Distributed Systems
COMP90015 2019 SM1
OS Support
Lectures by Aaron Harwood
© University of Melbourne 2019

Overview

- operating system layer
- protection
- processes and threads
- communication and invocation
- operating system architecture

Networking versus Distributed OS

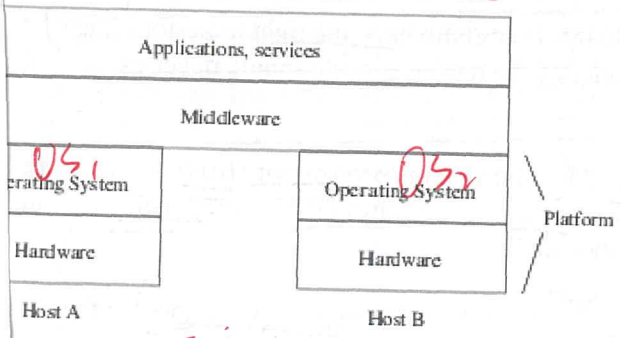
networked operating system provides support for networking operations. The users are generally expected make intelligent use of the network commands and operations that are provided. Each host remains tonomous in the sense that it can continue to operate when disconnected from the networking environment.

distributed operating system tries to abstract the network from the user and thereby remove the need for the er to specify how the networking commands and operations should be undertaken. This is sometimes ferred to as providing a *single system image*. Each host may not have everything that would be required to erate on its own, when disconnected from the network.

e term "single system image" is often discussed in the context of cluster computing. The MOSIX operating tem provides a single system image over a cluster and has proven to be quite successful.

general, consumer operating systems are networked. Many home consumers use only a single machine rway, and users in organizations tend to prefer autonomy.

figure depicts two different hosts, each with its own hardware and operating system, or platform, but ceptually supporting a consistent middleware that supports distributed applications and services.



igure is depicting network operating systems as opposed to distributed operating systems, since there is plied commonality between the OSes or the hardware. The operating system includes the kernel.

libraries and servers.

If the operating system is divided into *kernel* and *server* processes then they:

- *Encapsulate* resources on the host by providing a useful service interface for clients. Encapsulation hides details about the platform's internal operations; like its memory management and device operation.
- *Protect* resources from illegitimate access, from other users and other clients that are using resources on that host. Protection ensures that users cannot interfere with each other and that resources are not exhausted to the point of system failure.
- *Concurrently process* client requests, so that all clients receive service. Concurrency can be achieved by sharing time -- a fundamental resource -- called time sharing.

E.g. a client may allocate memory using a kernel system call, or it may discover an network address by using a server object. The means of accessing the encapsulated object is called an *invocation method*.

The core OS components are:

- *Process manager* -- Handles the creation of processes, which is a unit of resource management, encapsulating the basic resources of memory (address space) and processor time (threads).
- *Thread manager* -- Handles the creation, synchronization and scheduling of one or more threads for each process. Threads can be scheduled to receive processor time.
- *Communication manager* -- Handles interprocess communication, i.e. between threads from different processes. In some cases this can be across different hosts.
- *Memory manager* -- Handles the allocation and access to physical and virtual memory. Provides translation from virtual to physical memory and handles paging of memory.
- *Supervisor* -- Handles privileged operations, i.e. those that directly affect shared resources on the host e.g. to and from an I/O device. The supervisor is responsible for ensuring that host continues to provide proper service to each client.

In a number of systems that are emerging today, there is also a *hypervisor* that sits beneath the supervisor. The hypervisor allows a host to concurrently execute multiple kernels. There is typically multiple levels of virtual memory and additional overheads for managing access to I/O devices, etc.

7.3 Protection

Resources that encapsulate space, such as memory and files, typically are concerned with *read* and *write* operations. Protecting a resource requires ensuring that only legitimate read and write operations take place.

Legitimate operations are those carried out only by clients who have the right to perform them. A legitimate operation should also conform to resource policies of the host, e.g. a file should never exceed 1 GB in size o at most 100MB of memory can be allocated.

In some cases the resource may also be protected by giving it the property of *visible* versus *invisible*. A visib resource can be discovered by listing a directory contents or searching for it. An invisible resource should b known a priori to the client; it can be guessed though.

Resources that encapsulate time, i.e. processes, are concerned with *execute* operations. In this case a client may or may not have the right to create a process. Again, host based policies should be enforced.

Files in the UNIX operating system have the three fundamental access privileges of read, write and execute. The execute privilege is used on directories to determine if the contents can be listed or not; since a directory is not executable.

The kernel is that part of the operating system which assumes full access to the host's resources. The kernel begins execution soon after the host is powered up and continues to execute while the host is operational. The kernel has access to all resources and shares access to all other processes that executing on the host. To do this securely requires hardware support at the machine instruction level, which is supplied by the processor using two fundamental operating modes:

- *supervisor* mode -- instructions that execute while the processor is in supervisor (or privileged) mode are capable of accessing and controlling every resource on the host,
- *user* mode -- instructions that execute while the processor is in user (or unprivileged) mode are restricted, by the processor, to only those accesses defined or granted by the kernel.

Most processors have a register that determines whether the processor is operating in user or supervisor mode.

Before the kernel assigns processor time to a user process, it puts the processor into user mode.

A user process accesses a kernel resource using a *system call*. The system call is an *exception* that puts the processor into supervisor mode and returns control to the kernel.

Other kinds of exceptions can occur, such as when an illegal instruction has been issued, a resource is trying to be accessed that is not allowed, or when I/O is pending.

In theory, a well designed, error-free kernel, that uses proper hardware level security, cannot "loose control" of the host. Implementing such a kernel is a very challenging task.

On systems with supervisor mode there is typically a user, e.g. the superuser or root user, that can execute any desired processes in supervisor mode, which is necessary to maintain and administer the host. Such a user is a trusted user. Other users are typically limited to what processes they can execute in supervisor mode.

Of course, supervisor mode can only access and control every resource on the host if there is no active hypervisor.

Processes and threads

A process encapsulates the basic resources of memory and processor time. It also encapsulates other higher level resources.

Each process:

- has an address space and has some amount of allocated memory,
- consists of one or more threads that are given processor time, including thread synchronization and communication resources,
- higher-level resources like open files and windows.

Threads have equal access to the resources encapsulated within the process.

Resource sharing or interprocess communication is required for threads to access resources in other processes.

Threads can be created and destroyed dynamically
=> to maximum the degree of concurrent execution between operations

E.g. shared memory or socket communication.

The allocation of a resource does not come for "free". E.g., each process requires a process table entry and each open file requires a file descriptor to be recorded in memory.

Address spaces

Most operating systems allocate a *virtual* address space for each process. The virtual address space is typically byte addressable and on a 32 bit architecture will typically have 2^32 byte addresses.

The virtual address space can be divided into *regions* that are contiguous and do not overlap.

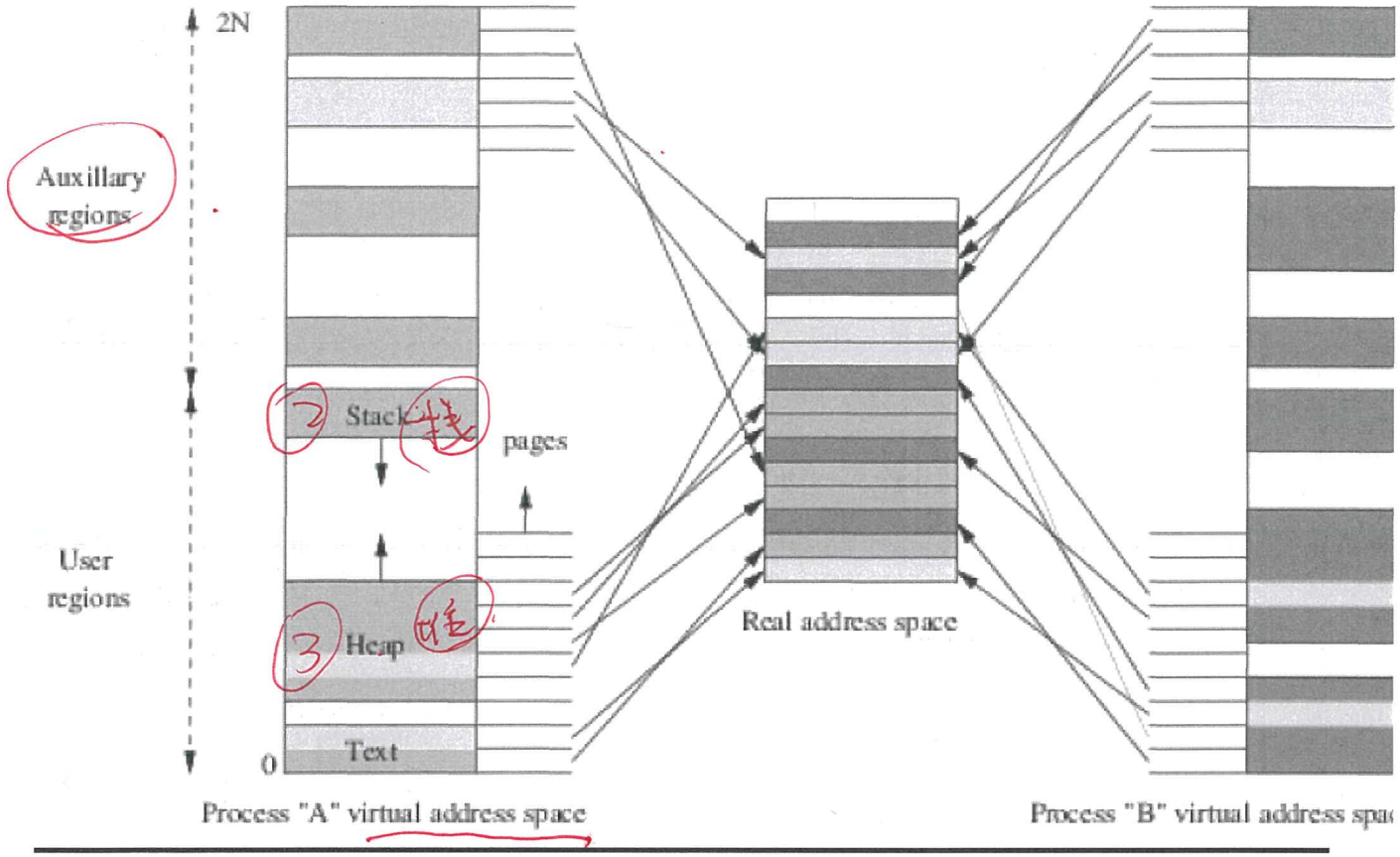
A *paged* virtual memory scheme divides the address space into fixed sized blocks that are either located in physical memory (RAM) or located in swap space on the hard disk drive.

A *page table* is used by the processor and operating system to map virtual addresses to real addresses. The page table also contains access control bits for each page that determine, among other things, the access privileges of the process on a per page basis.

The operating system manages the pages, swapping them into and out of memory, in response to process memory address accesses.

The size of memory address register in the CPU determines the size of the address space. While it is possible to have 2^64 bytes of addressable memory on a 64 bit architecture, in most cases the extra bits are used for other purposes.

Example virtual address space mappings



Shared memory

Two separate addresses spaces can share parts of real memory. This can be useful in a number of ways:

- *Libraries*: The binary code for a library can often be quite large and is the same for all processes that use it. A separate copy of the code in real memory for each process would waste real memory space. Since the code is the same and does not change, it is better to share the code.
- *Kernel*: The kernel maintains code and data that is often identical across all processes. It is also often located in the same virtual memory space. Again, sharing this code and data can be more efficient than having several copies.
- *Data sharing and communication*: When two processes want access to the same data or want to communicate then shared memory is a possible solution. The processes can arrange, by calling appropriate system functions, to share a region of memory for this purpose. The kernel and a process can also share data or communicate using this approach.

Examples of each of these shared memory uses are shown in the previous figure.

Creation of a new process

The operating system usually provides a way to create processes. In UNIX the `fork` system call is used to duplicate the caller's address space, creating a new address space for a new process. The new process is identical to the caller, apart from the return value of the fork system call is different in the caller. The caller is called the parent and the new process is called the child.

In UNIX a `exec` system call can be used to replace the caller's address space with a new address space for a new process that is named in the system call.

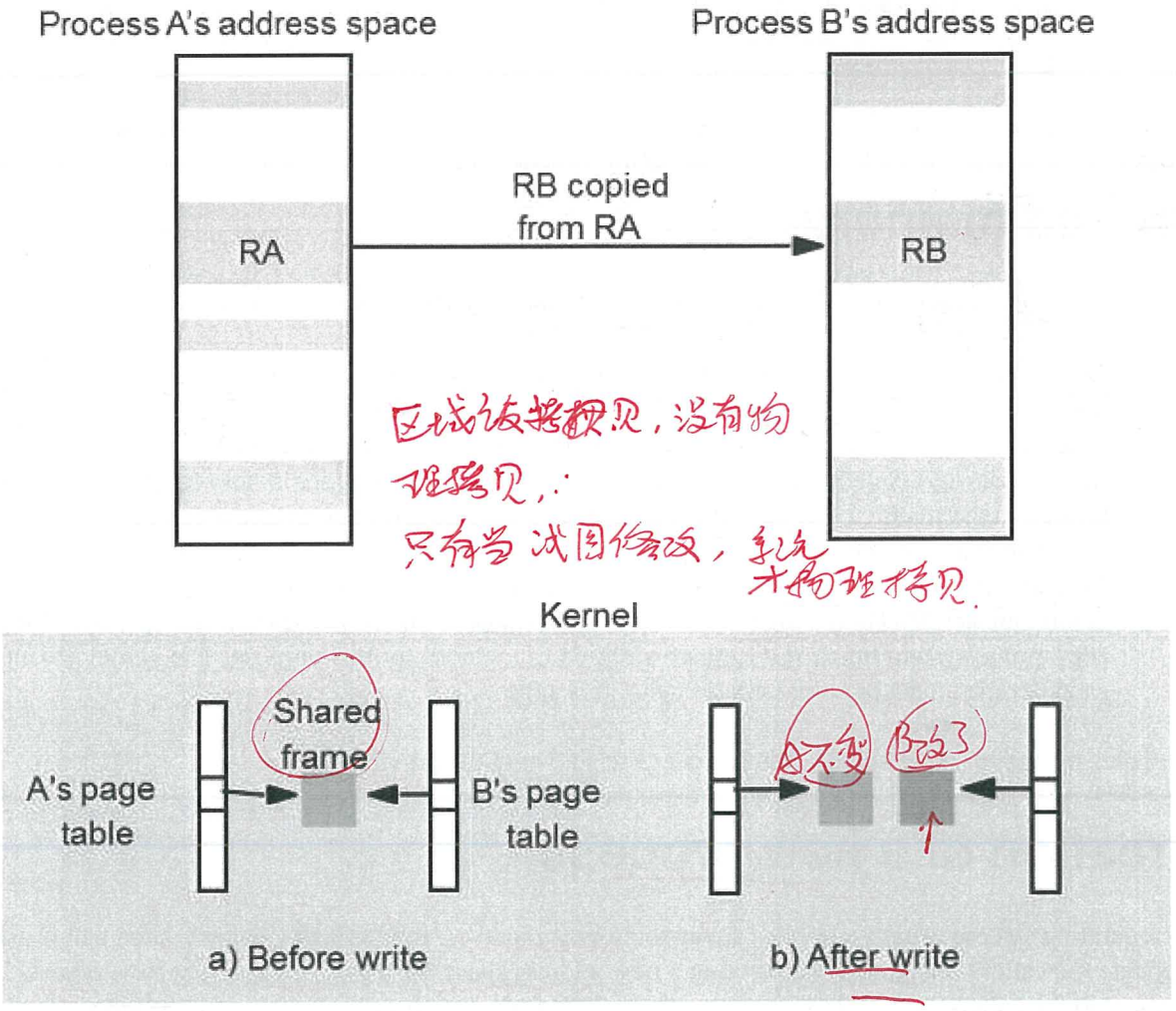
A combination of fork and exec allows new processes to be allocated.

Copy on write

When a new process is created using fork, the address space is copied. The new process' code is identical and is usually read-only so that it can be shared in real memory and no actual copying of memory bytes is required. This is faster and more efficient than making a copy.

However the data and other memory regions may or may not be read-only. If they are writable then the new process will need its own copy when it writes to them.

Copy on write is a technique that makes a copy of a memory region only when the new process actually writes to it. This saves time when allocating the new process and saves memory space since only what is required to be copied is actually copied.



New processes in a distributed system

In a distributed system there is a choice as to which host the new process will be created on. In a distributed operating system this choice would be made by the operating system.

The decision is largely a matter of policy and some categories are:

- *transfer policy* -- determines whether the new process is allocated locally or remotely.
- *location policy* -- determines which host, from a set of given hosts, the new process should be allocated on.

The policy is often transparent to the user and will attempt to take into account such things as relative load across hosts, interprocess communication, host architectures and specialized resources that processes may require.

When the user is programming for explicit parallelism or fault tolerance, then they may require a means for specifying process location. However, it is desirable to make these choices automatic as well.

Process location policies may be *static* or *adaptive*. Static policies do not take into account the current state of the distributed system. Adaptive policies receive feedback about the current state of the distributed system.

A *load manager* gathers information about the current state of the distributed system. Load managers may be:

- *centralized* -- a single load manager receives feedback from all other hosts in the system.
- *hierarchical* -- load managers are arranged in a tree where the internal nodes are load managers and the leaf nodes are hosts.
- *decentralized* -- there is typically a load manager for every host and load managers communicate with all other load managers directly.

In a *sender-initiated* or push policy, the local host is responsible for determining the remote host to allocate the process. In the *receiver-initiated* or pull policy, remote hosts advertises to other hosts that new processes should be allocated on it.

Sometimes it is more efficient to use push over pull and vice versa.

Process migration

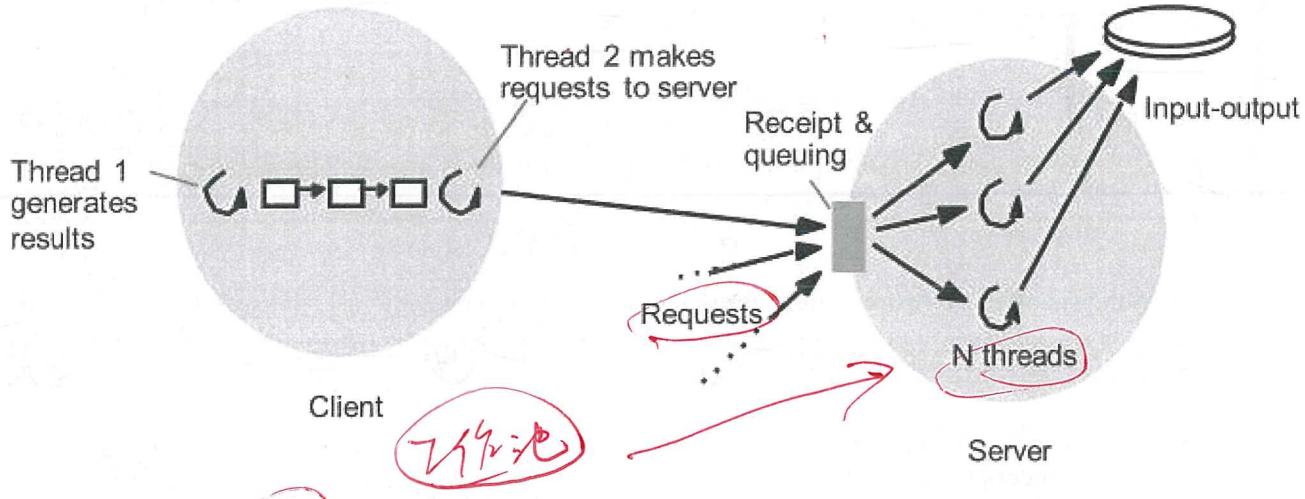
Processes can be *migrated* from one host to another by copying their address space. Depending on the platform and on the resources that are in current use by the process, process migration can be more or less difficult.

Process code is often CPU dependent, e.g. x86 versus SPARC. This can effectively prohibit migration. Ensuring that hosts are homogeneous can eliminate this problem. Using virtual machines can also help, by avoiding CPU dependent instructions.

Even if the host platforms are such that a process can migrate, the process may be using host resources such as open files and sockets that further complicates the migration.

The Condor library is used by the Department's P2P research group to migrate Linux processes in an x86 environment.

Threads



Typically there is significant use of threads in a distributed system.

Performance bottleneck

Consider a server than handles client requests. Handling the request requires a disk access of 8ms and 2ms of processor time. A process with a single thread takes 10ms of time and the server can complete 100 requests/second.

If we use two threads, where each thread independently handles a request then while the first thread is waiting for the disk access to complete the second thread is executing on the processor. In general if we have many threads then the server becomes bottlenecked at the disk drive and a request can be completed every 8ms which is 125 requests/second.

Consider when disk accesses are cached with a 75% hit rate and an increase in processor time to 2.5ms (the increase is due to the cache access). The disk access is now completed in 0.75x0 + 0.25x8 = 2ms. Hence the requests are bottlenecked at the processor and the maximum rate is 1000/2.5=400 requests/second.

Some multi-processor operating systems do not schedule threads of a processes on different processors; though most do.

Worker pool architecture

Creating a new thread incurs some overhead that can quickly become the bottleneck in a server system. In this case it is preferable to create threads in advance.

It is okay to have a single I/O thread to receive the requests since the I/O is the bottleneck.

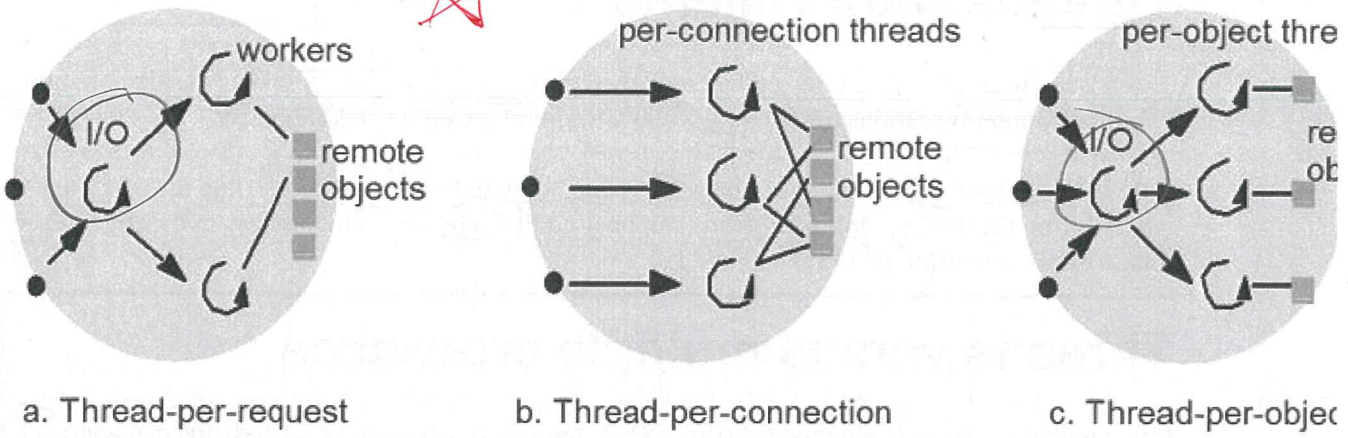
In the *worker pool* architecture the server creates a fixed number of threads called a worker pool. As requests arrive at the server, they are put into a queue by the I/O thread and from there assigned to the next available worker thread.

Request priority can be handled by using a queue for each kind of priority. Worker threads can be assigned to high priority requests before low priority requests.

If the number of workers in a pool is too few too handle the rate of requests then a bottleneck forms at the queue. Also, the queue is shared and this leads to an overhead.

The Common Object Request Broker Architecture uses the worker pool architecture.

Alternative Threading



Thread-per-request architecture

In the *thread-per-request* architecture a separate thread is created by the I/O thread for each request and the thread is deallocated when the request is finished.

This allows as many threads as requests to exist in the system and avoids accessing a shared queue. Potential parallelism can be maximized.

However, thread allocation and deallocation incurs overheads as mentioned earlier. Also, as the number of threads increases then the advantages from parallelism decreases (for a fixed number of processors) and the overhead incurred in context switching between threads can outweigh the benefit. Managing large numbers of threads can be more expensive than managing a large queue.

Thread-per-connection architecture

A single client may make several requests. In the previous architectures, each request is treated independently of the client connection.

In the *thread-per-connection* architecture, a separate thread is allocated for each *connection* rather than for each *request*. This can reduce the overall number of threads as compared to the thread-per-request architecture.

The client can make several requests over the single connection.

Thread-per-object architecture

In the *thread-per-object* architecture, a worker thread is associated for each remote object or resource that is being accessed. An I/O thread receives requests and queues them for each worker thread.

The thread-per-connection and thread-per-object architectures generally use less threads, however bottlenecks can still occur. For example the thread-per-connection can leave a client waiting on many requests, even though there are idle processors at the server; similarly for the thread-per-object architecture.

Threads within clients

Threads are useful within clients in the case when the request to the server takes considerable time. Also, communication invocations often block while the communication is taking place.

E.g., in a web browser the user can continue to interact with the current page while the next page is being fetched from the server. Multiple threads can be used to fetch each of the images in the page, where each image may come from a different server.

Threads versus multiple processes

It is possible to use multiple processes instead of multiple threads. However the threaded approach has the advantages of being cheaper to allocate/deallocate and of being easy to share resources via shared memory.

One study showed that creating a new process took about 11ms while creating a new thread took 1ms. While these times may be decreasing as as technology improves, the relative difference is likely to remain.

This is because, e.g., processes require a new address space which leads to a new page table, while threads require in comparison only a new processor context.

If the kernel does not schedule threads across processors then it can be preferable to use multiple processes rather than multiple threads.

In some Java implementations, the Java threads are not scheduled across different processors. They are in Solaris' Java implementation.

Context switching between threads is also cheaper than between processes because of cache behavior concerned with address spaces. Some processors are optimized to switch between threads (hyper-threading) and so this can lead to greater advantages for the threading model; though it has caveats as well.

A *processor context* comprises the values of the processor registers such as the program counter, the address space identifier and the processor protection mode (supervisor or user).

Context switching involves saving the processor's original state and loading the new state. When changing from a user context to a kernel context it also involves changing the protection mode which is a called a *domain transition*.

One study showed that context switching between processes took 1.8ms while switching between threads belonging to the same process took 0.4ms. Again, it is the relative difference that is of interest here.

A danger with the use of threads is that their memory space is not protected from each other and so an errant thread can easily corrupt the data of other threads.

Thread programming

Thread programming is largely concerned with the study of concurrency, including:

- race condition,
- critical section,
- monitor,
- condition variable, and
- semaphore.

In Java, the `Thread` class is used to implement threads.

Typically, each OS provides its own kind of thread model. Recently the POSIX thread model, known as pthreads, has become standard.

For parallel programming there is the OpenMP programming model. It provides an abstraction on top of the threading model that is particularly suited for parallel processing but that can also be used for general processing.

Java thread methods

- `Thread(ThreadGroup group, Runnable target, String name)` -- Creates a new thread in the `SUSPENDED` state, which will belong to `group` and be identified as `name`; the thread will execute the `run()` method of `target`.
- `setPriority(int newPriority), getPriority()` -- Set and return the thread's priority.
- `run()` -- A thread executes the `run()` method of its target object, if it has one, and otherwise its own `run()` method.
- `start()` -- Change the state of the thread from `SUSPENDED` to `RUNNABLE`.
- `sleep(int millisecs)` -- Cause the thread to enter the `SUSPENDED` state for the specified time.
- `yield()` -- Enter the `READY` state and invoke the scheduler.
- `destroy()` -- Destroy the thread.

Thread lifetime

A new thread is created on the Java virtual machine (JVM) as its creator.

It is created in the `SUSPENDED` state.

It is made `RUNNABLE` using the `start()` method and executes the `run()` method (either designated or its own).

Threads can be assigned priorities for scheduling purposes and can be managed in groups.

Groups provide protection of threads within a group from threads in other groups.

synchronization

Java thread synchronization

- `thread.join(int millisecs)` -- Blocks the calling thread for up to the specified time until `thread` has terminated.
- `thread.interrupt()` -- Interrupts `thread`: causes it to return from a blocking method call such as `sleep()`.
- `object.wait(long millisecs, int nanosecs)` -- Blocks the calling thread until a call made to `notify()` or `notifyAll()` on `object` wakes the thread, or the thread is interrupted, or the specified time has elapsed.
- `object.notify()`, `object.notifyAll()` -- Wakes, respectively, one or all of any threads that have called `wait()` on `object`.

Consider a shared function to generate unique identifiers:

```
int greatest=0;
Stack idStack=new Stack();
public int generateUniqueID(){
   if(!idStack.empty()){
      Integer top =
         (Integer) idStack.pop();
      while(top.intValue()>greatest)
         top = (Integer) idStack.pop();
      return top.intValue();
   }
   greatest++;
   return greatest-1;
}
```

This code doesn't work when several threads call it concurrently.

Consider when there is exactly one element on the stack. The condition to check whether the stack is empty may be tested concurrently by two or more threads. All threads will assume that the stack contains a value to be popped and will attempt to pop it. In fact there are a lot of problems with executing this code concurrently.

One way to avoid these problems is to allow only one thread to be executing the function at a time. This eliminates concurrency for this function which reduces the parallelism but avoids the concurrent access problem.

Using a stack is a good way to keep track of which integers have been returned for reuse. The value for `greatest` is only decremented when the `(greatest-1)` identifier is returned.

The POSIX standard defines a `pthread_mutex_t` data type and functions such as `pthread_mutex_init()`, `pthread_mutex_lock()`, `pthread_mutex_unlock()` and `pthread_mutex_trylock()` for blocking and non-blocking mutual exclusion operation.

Among other things, Java provides the `synchronized` keyword to indicate methods for which only one thread may be executing at any one time.

Monitors allow a higher level thread synchronization control by providing `wait` and `signal` primitives.

Mutual exclusion can lead to `deadlock`, `livelock` and `starvation`.

Thread scheduling

Some thread scheduling is *preemptive*. In this case a running thread is suspended at any time, usually periodically, to allow processor time for other threads.

Other the thread scheduling is *non-preemptive*. In this case a running thread will continue to receive processor time until the thread *yields* control back to the thread scheduler.

Non-preemptive scheduling has the advantage that concurrency issues, such as mutual exclusion to a shared variable, are greatly simplified.

However, non-preemptive scheduling does not guarantee that other threads will receive processor time since an errant thread can run for an arbitrarily long time before yielding. This can have negative effects on performance and usability.

Real time scheduling of threads is not support by vanilla Java. Real time imposes another set of constraints on thread scheduling.

Thread implementation

Depending on the kernel, the threads of a process may or may not be schedulable across different processors.

Some kernels provide system calls only to allocate processes. Some kernels provide system calls to allocate threads as well; they can be call *kernel threads*. This can depend on whether the kernel itself is threaded. Most modern OSes have threaded kernels.

If the kernel does not provide thread allocation (and even if it does), it is possible for the user to manage *user threads* within a process.

In the case of user threads, the kernel gives processor time to the process and the process is responsible for scheduling the threads.

In ANSI C, user threads can be implemented using library functions such as `setjmp()` and `longjmp()`.

In UNIX the `ps` command lists the currently running processes. Usually, the `-M` option lists threads. Some kernels allocate separate process numbers to each thread. Some kernels do not.

User versus kernel threads

1. User threads within a process cannot take advantage of multiple processors.
2. A user thread that causes a page fault will block the entire process and hence all of the threads in that process.
3. User threads within different processes cannot be scheduled according to a single scheme of relative prioritization.

1. Context switching between user threads can be faster than between kernel threads.
2. A user thread scheduler can be customized by the user, to be specific for a given application.

Usually the kernel places limits on how many kernel threads can be allocated, a user thread scheduler can usually allocate more threads than this.

It is possible to combine kernel level threads with user level threads.

Communication and invocation

An invocation such as a remote method invocation, remote procedure call or event notification is intended to bring about an operation on a resource in a different address space.

- What communication primitives are available?
- Which protocols are supported and how open are they?
- What is done to make the communication efficient?
- What additional support is provided, e.g. for high-latency communication and issues such as disconnection?

The kernel can provide communication primitives such as TCP and UDP, and it can also provide higher level primitives. In most cases the higher level primitives are provided by middleware because it become too complex to develop them into the kernel and because standards are not widely accepted at the higher level.

Protocols and openness

Using open protocols, as opposed to closed or proprietary protocols, facilitates interoperation between middleware implementations on different systems.

Experience has shown that kernels which implement and require their own network protocols do not become popular.

A design choice of some new kernels is to leave the implementation of networking protocols to servers. Hence a choice of networking protocol can be more easily made.

The kernel is still required to provide device drivers for new networking devices such as infrared and BlueTooth. Either the kernel can transparently select the appropriate device driver or middleware should be able to dynamically select the appropriate driver. In either case, standard protocols such as TCP and UDP allow the middleware and application to make use of the new devices without significantly changes.

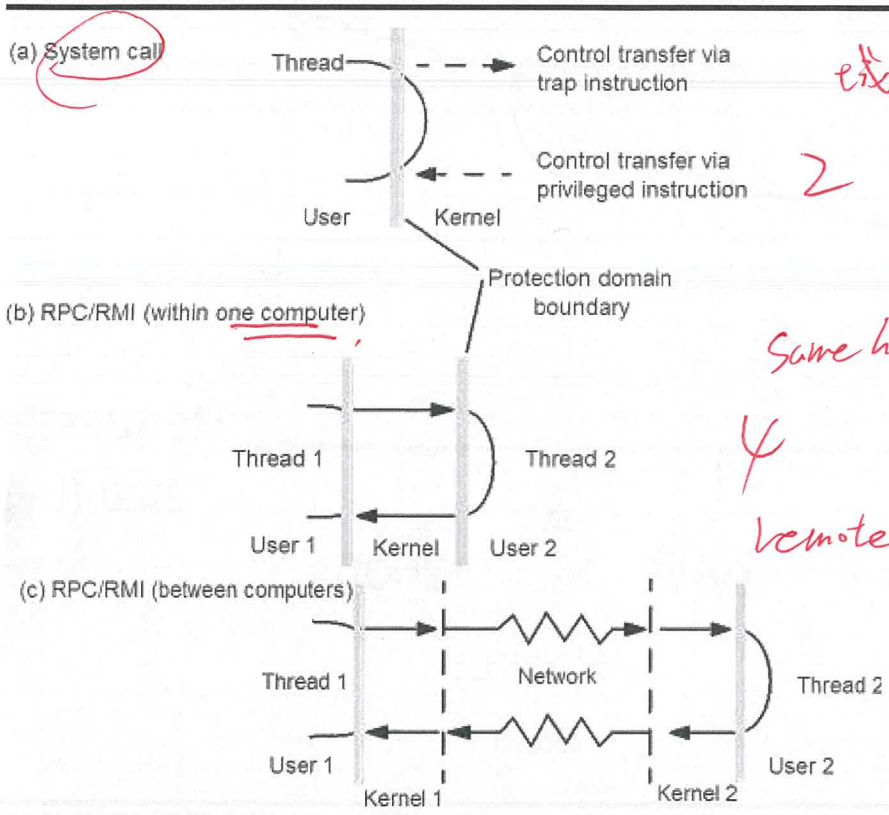
Invocation performance

The performance of an invocation can be categorized into three kinds, depending on what is required to invoke the resource:

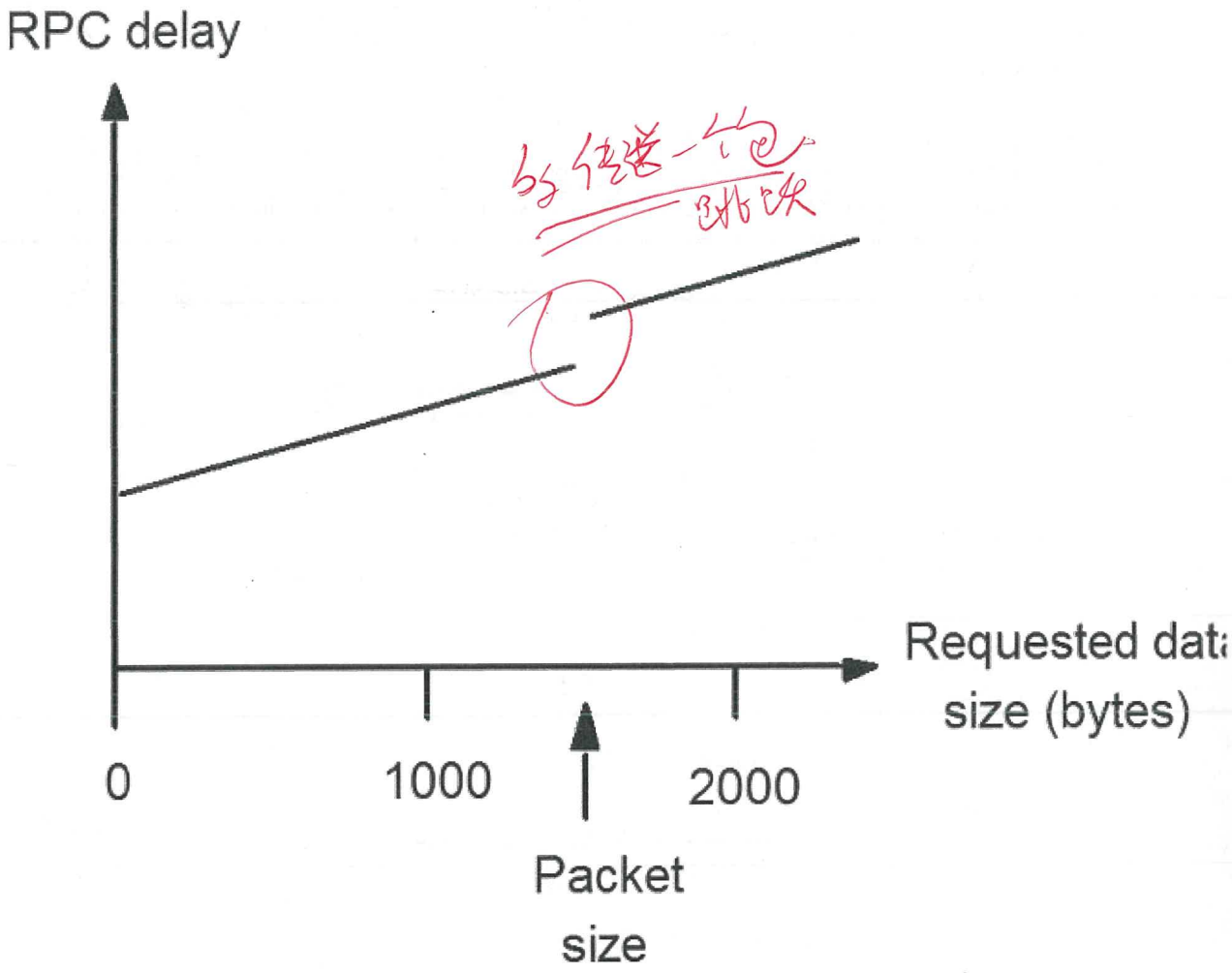
- User space procedure -- minimal overhead, allocating stack space.
- System call -- if a system call is required then the overhead is characterized by a domain transition two and from the kernel.
- Interprocess on the same host -- in this case there is a domain transition to the kernel, to the other process, back to the kernel and back to the calling process.
- Interprocess on a remote host -- this case includes the same overheads as the previous case, plus the overhead of network communication between the two kernels.

One study showed that the time for a null RPC between user processes on two 500MHz PCs across a 100Mbps LAN is in the order of tenths of a millisecond. By comparison an equivalent user space procedure call takes a small fraction of a microsecond.

Consider printing lots of information to a log file. Using several calls to the system print function is less efficient than buffering the information in user space and making a single call to the system print function. In part this is also due to combining several disk accesses into a a smaller number of larger accesses.



RPC Delay versus size



The following are the main factors contributing to delay for RMI, apart from actual network delay:

- marshalling -- copying and converting data becomes significant as the amount of data grows.
- data copying -- after marshalling, the data is typically copied several times, from user to kernel space, and to different layers of the communication subsystem.
- packet initialization -- protocols headers and checksums take time, the cost is proportional to the size of the data.
- thread scheduling and context switching -- system calls and server threads.
- waiting for acknowledgments -- above the network level acknowledgments.

Communication via shared memory

Shared memory can be used to communicate between user processes and between a user process and the kernel.

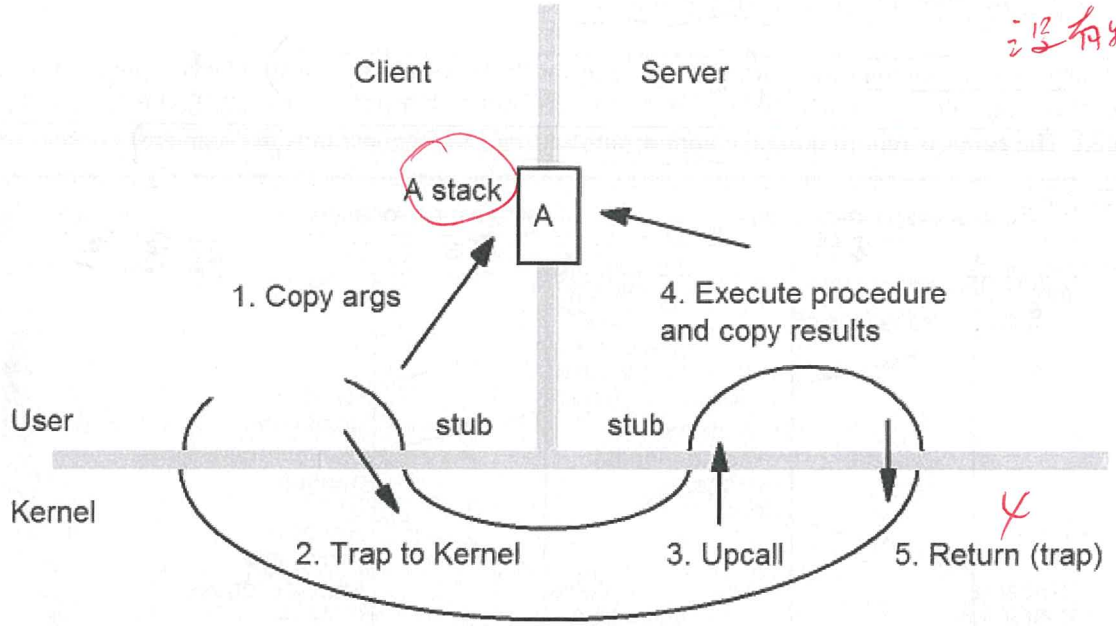
Data is communicated by writing to and reading from the shared memory, like a "whiteboard".

In this case, when communicating between user processes the data is not copied to and from kernel space.

However the processes will typically need to use some synchronization mechanism to ensure that communication is deterministic.

In UNIX, a shared memory area can be obtained using `shmget()`, the area can be attached to a process using `shmat()`, detached using `shmdt()` and controlled (e.g. specifying read/write permissions) using `shmctl()`. The `shmget()` is a system call that returns an identifier and the identifier must be communicated to other processes. Thus there is some overhead in setting up a shared memory area.

Lightweight RPC



Choice of protocol

Connection-oriented protocols like TCP are often used when the client and server are to exchange information in a single session for a reasonable length of time; e.g. telnet or ssh. These protocols can suffer if the end-points are changing their identity, e.g. if one of the end-points is mobile and roaming from one IP address to another, or e.g. if a DHCP based wireless base station experiences a lot of contention and the clients to the base station are continually having their IP address change.

Connection-less protocols like UDP are often used for request-reply applications, that do not require a session for any length of time. E.g., finding the time from a time server. Because these protocols often have less overhead, they are also used for applications that require low latency, e.g. in streaming media and online games.

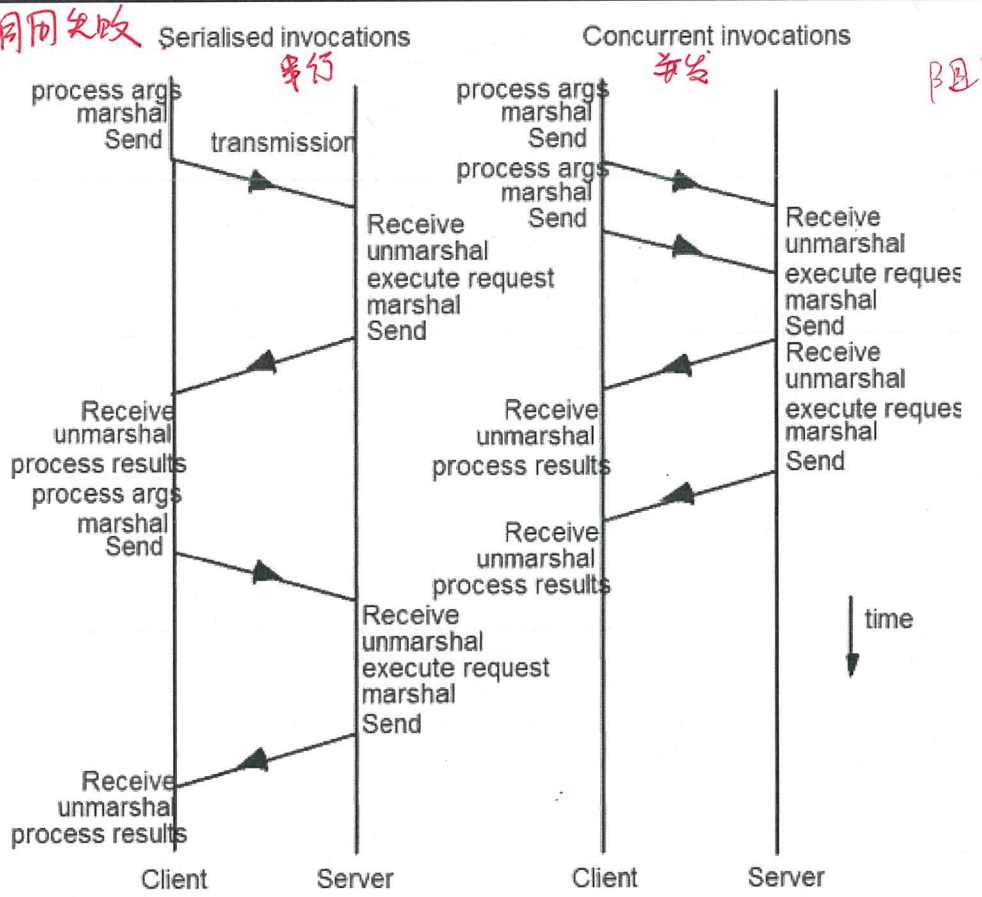
In many cases, cross address space communication happens between processes on the same host and so it is preferable to (transparently) choose a communication mechanism (such as using shared memory) that is more efficient. Lightweight RPC is an example of this.

Concurrent and asynchronous operation

In many applications it is possible to have a number of outstanding or *concurrent* requests. In other words it is not necessary to wait for the response to a request before moving on to another request. This was exemplified in the use of threads for the client, where e.g. several requests to different web servers could be undertaken concurrently.

The telnet client/server is an example of *asynchronous* operation. In this case, whenever a key is typed at the keyboard the client sends the key to the server. Whenever output is available from the server it is sent to the client and received data from the server is printed on the terminal by the client whenever it arrives. Sends and receives are not synchronized in any particular way.

An asynchronous invocation returns without waiting for the invocation request to be completed. Either the caller must periodically check whether the request has completed or the caller is notified when the request has completed. The caller is required to take appropriate action if the request fails to complete.



Persistent asynchronous invocations

With usual networking circumstances it is possible to automatically retry a nonresponsive request after a timeout, retrying some number of times before the request is considered to have failed. Usually the timeout period is some number of seconds and is sufficient for example when making a request from a client on the Internet to a server on the Internet.

① invocation 放到 queue，当网络通畅，向远程发给服务器
② server返回的结果放到 "client mailbox"；直到client get the result.

However, with the use of mobile devices that can experience excessively long disconnection times it is preferable to handle asynchronous invocations in a different way. A *persistent asynchronous invocation* is placed in a queue at the client and attempts are made to complete the request as the client roams from network to network. At the server side, the response to the request is put into a "client mailbox" and the client is required to retrieve the response when it can.

Persistent invocations allow the user to select which kind of network (e.g. GSM or Internet) will be used to service the request. Queued RPC is an example of this.

Operating system architecture

An open distributed system should make it possible to:

仅运行在系统承载特定角色的软件.

- Run only that system software that is specifically required by the hosts hardware, e.g. specific software for a mobile phone or personal digital assistant. This leaves as much resources as possible available for user applications.
- Allow the software (and the host) implementing any particular service to be changed independently of other facilities.
- Allow for alternatives of the same service to be provided, when this is required to suit different users or applications.
- Introduce new services without harming the integrity of existing ones.

实现特定服务的软件可以独立地被替换

为不同用户，允许同一服务的不同实现.

不影响已有系统，可加入新的服务.

Monolithic and Micro kernel design

整体内核

The UNIX operating system is described as having a *monolithic kernel*. In this case there is significantly detailed and diverse functionality supplied by the kernel. Changing kernel functionality is more difficult than changing a server functionality. Same implementations use *kernel modules* to load and unload functionality into the kernel.

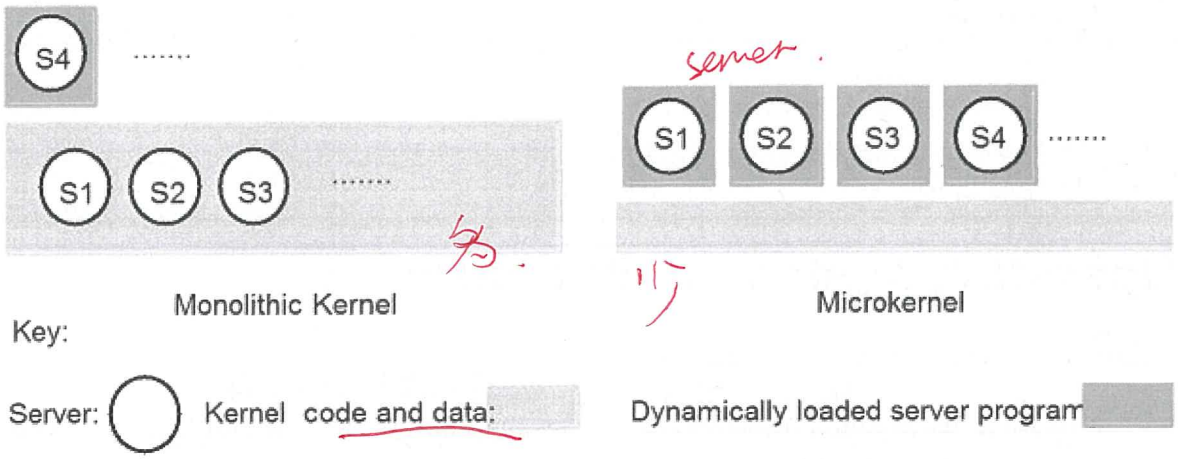
In contrast, the Mach operating system is described as a *micro kernel*. In this case the kernel provides only the fundamental functionality such as address space management, threads and local interprocess communication. All other functionality is provided by servers.

只提供最基本的，剩下的 functionality 由 servers 来做

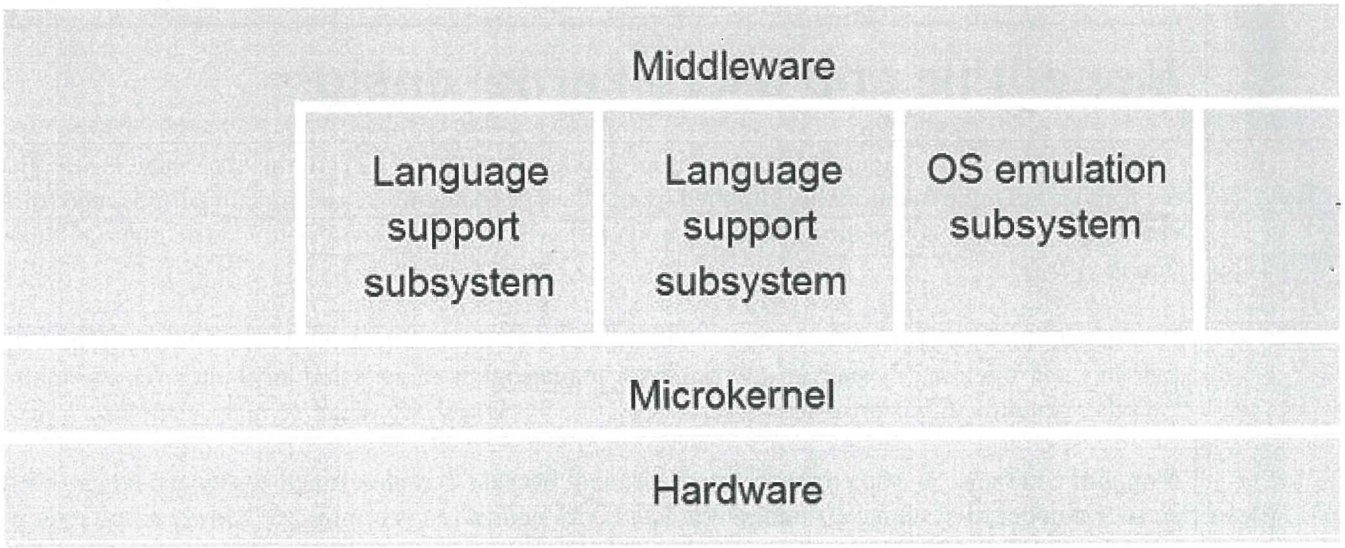
Monolithic kernels can sometimes be more efficient because complex functionality can be provided through a smaller number of system calls and interprocess communication is minimized. However micro kernels allow a greater extensibility and have a better ability to enforce modularity behind memory protection boundaries. A small kernel is also more likely to be free of bugs.

monolithic : ① smaller number of system calls. ② interprocess communication is minimized

micro kernel : ① greater extensibility ② better ability to enforce modularity.



Role of Microkernel



The microkernel supports middleware via subsystems

Emulation and virtualization

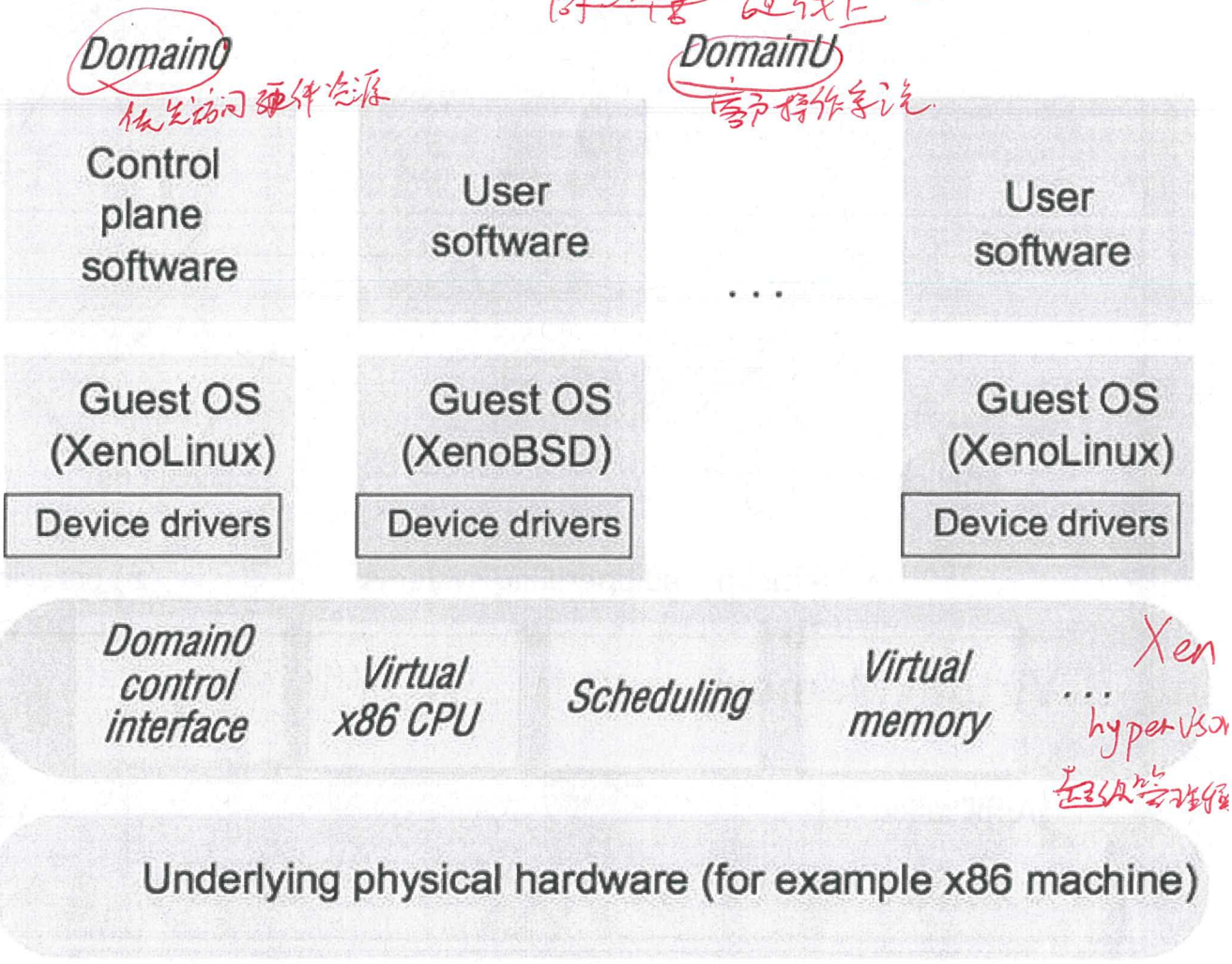
The adoption of microkernels is hindered in part because they do not run software that a vast majority of computer users want to use. Microkernels can use binary *emulation* techniques, e.g. emulating another operating system like UNIX, to overcome this. The Mach OS emulates both UNIX and OS/2.

Virtualization can also be used and can be achieved in different ways. Virtualization can be used to run multiple instances of virtual machines on a single real machine. The virtual machines are then capable of running different kernels. Another approach is to implement an OS' application programming interface, which

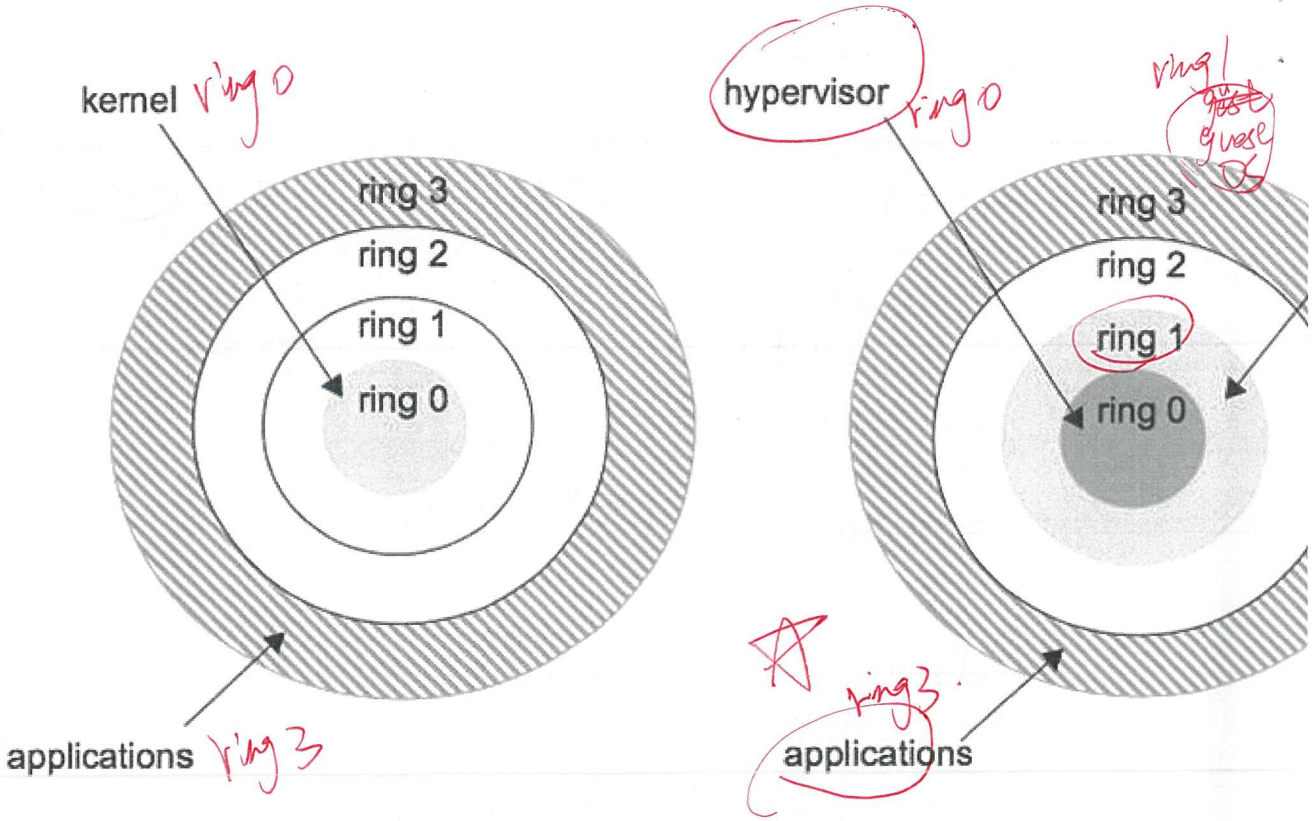
is what the UNIX Wine program does for Windows; i.e. it implements the Win32 API on UNIX.

Today, two prominent research systems that use virtualization are *Xen* and *PlanetLab*.

Xen Architecture



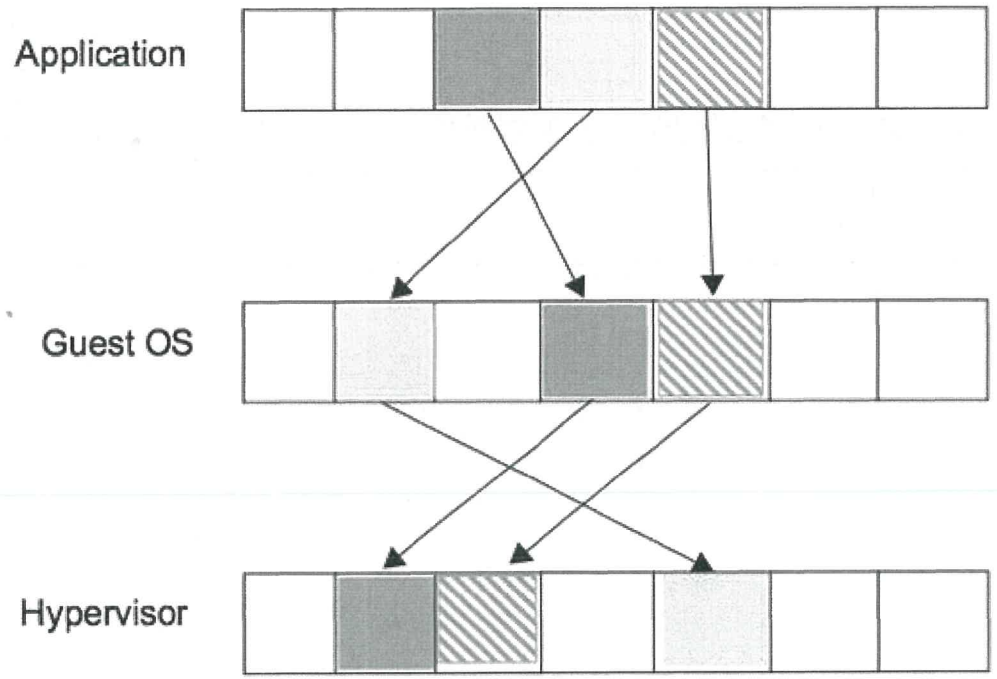
Privileges



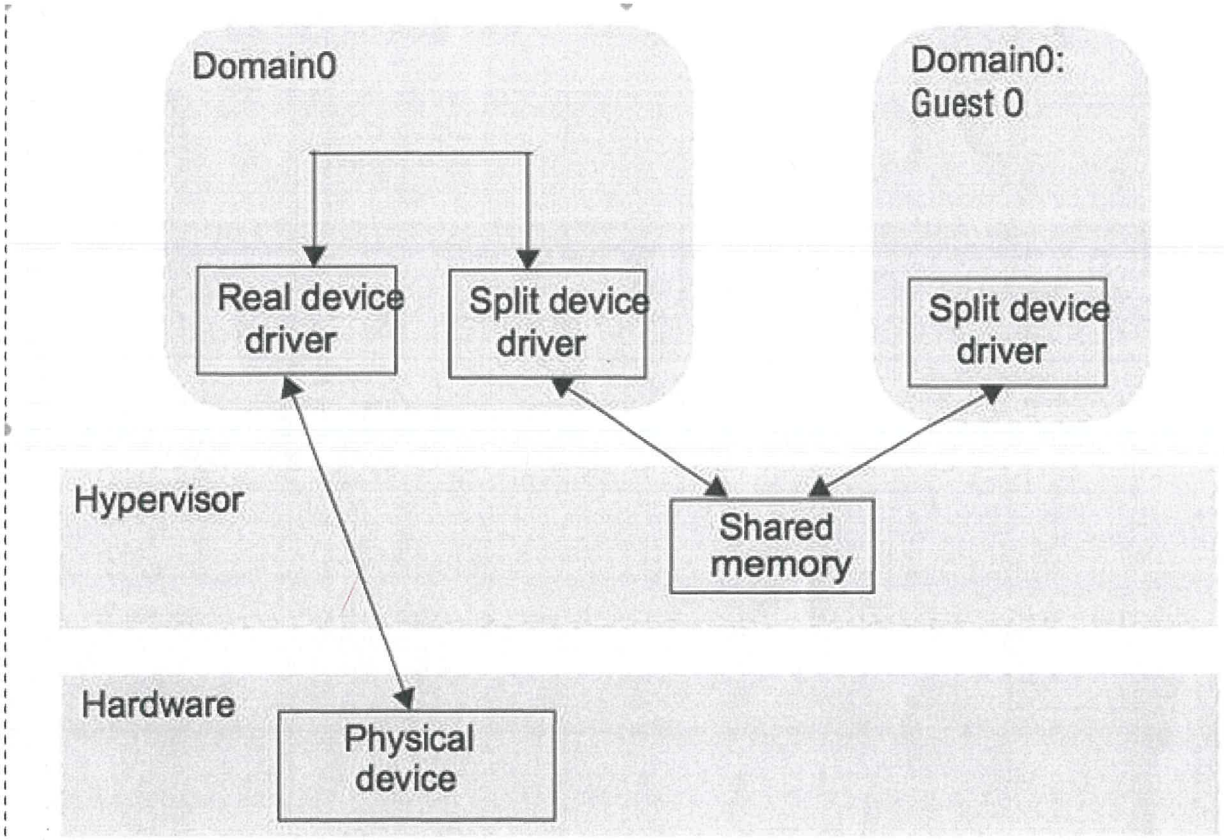
a) kernel-based operating systems

b) paravirtualization in X

Virtual Memory



Split Device Drivers



Distributed Systems
COMP90015 2019 SM1
Name Services
Lectures by Aaron Harwood
© University of Melbourne 2019

Names, addresses and other attributes

Names are used to refer to a wide variety of resources such as computers, services, remote objects and files, as well as to users. A name is needed to request a computer system to act upon a specific resource chosen out of many.

Processes need to be able to name resources in order to share them. Users need to be able to name each other in order to (privately or directly) communicate.

Communication may be possible without naming, as in the case of broadcasting. However, broadcasting is not private or directed.

In some cases a description that includes *attributes* of the resource can be used to uniquely identify it. In some cases a client requires a service but does not have a preference for a particular entity to provide that service.

An *address* is an attribute of an object. An address cannot be used as a name because the object may change its address.

Filenames such as `/etc/passwd` and URLs such as `http://www.cdk4.net/` are examples of *human-readable* names. An *identifier* is a name that is usually not human-readable, such as remote object references and NFS file handles. Identifiers can be more efficiently stored and processed by software.

A *pure name* contains no information about the object itself. Whereas a non pure name contains some information about the object; usually some kind of address information.

A pure name must be looked up, to obtain an address, before the named resource can be accessed. E.g., names of people are pure names.

An extreme example of a non pure name is an address. E.g. a user's email address is often used as a name for the user. However if the user changes their email address then the "name" is no longer correct.

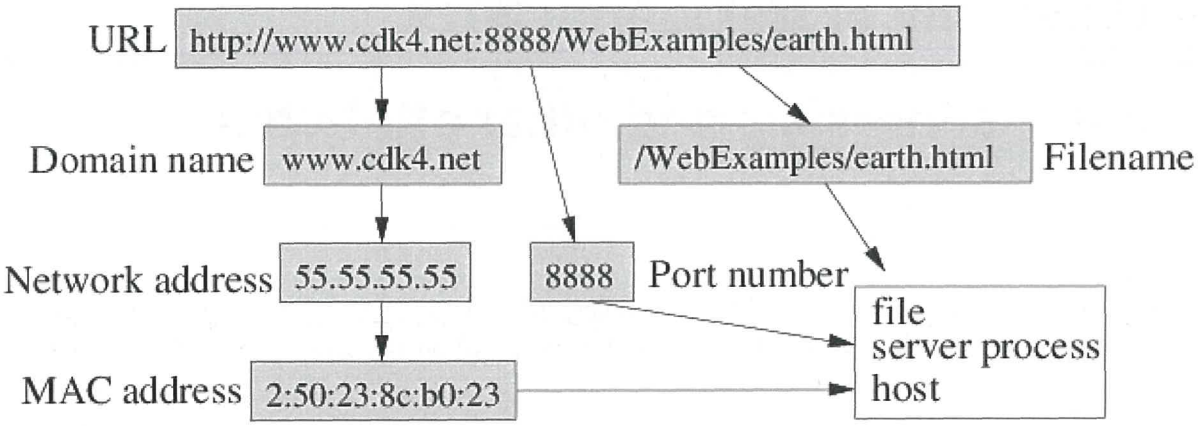
A name is *resolved* when it is translated into data about the named resource or object.

The association between a name and an object is called a *binding*.

In general, names are bound to attributes of the named objects and a common attribute is the address of the object.

- Domain Name System (DNS) maps human readable domain names to IP addresses and other kinds of attributes such as the type of host (e.g. mail server) and time for which the DNS entry remains valid.
- The X500 directory service can be used to map a person's name onto attributes including their email address and telephone number.

Example name



Names and services

Many names have meaning only to the service that creates the name, i.e. they have only *local significance*. When a service allocates a resource it also generates a unique name and a client to the service needs to supply the name in order to access the associated resource.

In some cases the client can specify the desired name for a new resource to the service. E.g., users will specify what email (user) name they want to use. The service needs to ensure that the names are locally unique. Along with a unique domain name, the email address is unique. There may be one or more hosts that maintain the service; with e.g. the domain name mapping to the host that is least loaded at the time of access.

Services sometimes need to cooperate in order to have name consistency. E.g., one attribute of a file is the owner which is a unique ID on the server. A login name resolves to a unique ID for that user. For the owner to be recognized by a client in NFS the client must ensure that user login names resolve to the same unique IDs as on the server.

There are other services that support this such as Lightweight Directory Access Protocol (LDAP).

Uniform Resource Identifiers

Uniform Resource Identifiers (URIs) are concerned with identifying resources on the Web, and other Internet resources such as electronic mailboxes.

URIs are intended to allow a generic way of specifying the identifier so as to make it easy for common software to process the identifier. This allows new types of identifiers to be readily introduced and for existing identifiers to be used by a wide variety of different software and services.

Very basically, the first part of a URI contains a short text mnemonic for the kind of resource being named, followed by a colon, e.g. `http:`, `widget:` or `tel:`. The remaining portion of the URI must be interpreted

according to the URI syntax which allows for local rules concerning the resource.

A URL is a URI. URLs provide ways to locate the resource being named. They clearly suffer if the resource has since changed its name (e.g. broken links in the Web).

Uniform Resource Names (URNs) are URIs that are used as pure resource names rather than locators. An URN requires a resolution service or name service to translate the URN into an actual address to find the named resource.

The URI prefix `urn:` has been allocated for URNs, e.g. `urn:ISBN:0-201-62433-8` identifies a resource (book) by the ISBN number, but there are many kinds of different URI prefixes in use today. E.g. for referring to documents:

- `doi:10.1007/s10707-005-4887-8` where the lookup service is `http://dx.doi.org/10.1007/s10707-005-4887-8` and this resolves to `http://www.springerlink.com/content/c250mn1u2m7n5586/` and in turn this refers to a document, "Building and Querying a P2P Virtual World".
- `oai:arXiv.org:cs/0605057` where the lookup service is `http://arxiv.org/abs/cs/0605057` and this resolves to a document. In this case the `oai` or Open Archives Initiative resource name includes the resolver service domain name.

Name Services

The major operation of a name service is to resolve a name, i.e to lookup the attributes that are bound to the name.

Name management is separated from other services largely because of the openness of distributed systems, which brings the following motivations:

- *Unification*: Resources managed by different services use the same naming scheme, as in the case of URIs.
- *Integration*: To share resources that were created in different administrative domains requires naming those resources. Without a common naming service, the administrative domains may use entirely different name formats.

Name services have become more important as the size of distributed systems has grown.

Goals of the Global Name Service

- To handle an essentially arbitrary number of names and to serve an arbitrary number of administrative organizations.
- A long lifetime, over many changes to the names and the system.
- High availability, dependent services stop working if the name server is unavailable.
- Fault isolation, local failures do not cause the entire service to fail.
- Tolerance of mistrust, a large open system cannot have any component that is trusted by all of the clients in the system.

Name spaces

A *name space* defines the set of names that are valid for a given service. The service will only lookup a valid name and may or may not resolve the name (the name may be unbound). E.g., "Two" is not a valid name for a UNIX process but "2" is.

Names may have a structure that is hierarchical, like a UNIX filename, or otherwise they are flat as in a randomly chosen integer identifier.

Structured names allow the lookup procedure to be more efficient and allows the name to incorporate semantics about the resource.

Names may be unbounded in length or fixed length. Unbounded length names allow the name to be whatever is required (e.g. by the user). Fixed length names are easier to store and process. E.g. consider the differences between a name string of arbitrary length and a 32 bit identifier.

An *alias* is a name that is typically substituted by the name service for another name that is harder to remember. In other cases the alias is a common name that allows ease of access to a resource which is managed using some other name. E.g. `www.example.com` may be an alias for `fred.example.com`, and later change to `alice.example.com` for management purposes. The alias does not change though. This provides transparency.

A *naming domain* is a name space for which there exists a single overall administrative authority for assigning names within it.

Administrative authority is usually delegated by dividing a domain into sub-domains. Each sub-domain shares a common part of the overall name in the name space.

Name resolution

Name resolution is in general an iterative process. A name either resolves to a set of primitive attributes or it resolves to another name.

The use of aliases make it possible for resolution cycles to occur and the potential for the resolution process to never terminate. Two solutions to overcome this include:

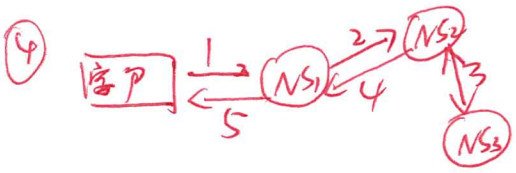
- Abandon the resolution process after some number of iterations.
- Require administrators to ensure that no cycles occur.

Distribution and navigation

Any name service that stores a very large database and is used by a large population will not store all of its naming information on a single server computer.

Bottlenecks include network I/O and server reliability.

Heavily used name services can use replication to increase availability.



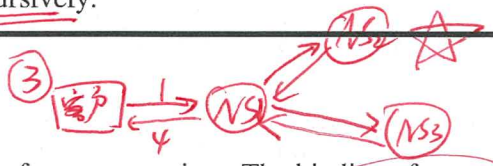
When name service authority is delegated then service distribution is also naturally distributed over the delegated authorities. In other words name service data is usually distributed in terms of domain ownership.

When the name service is distributed then a single server may not be able to resolve the name. The resolve request may need to propagate from one server to another, referred to as *navigation*.

The client name resolution software carries out navigation, e.g. on behalf of the application that requests for a name to be resolved. Approaches to navigation can be categorized in different ways:

- *iterative navigation* -- The client makes the request at different servers one at a time. The order of servers visited is usually in terms of domain hierarchy. Always starting at the root server would put excessive load on the root.
- *multicast navigation* -- The client multicasts the request to the group (or a subset) of name servers. Only the server that holds the named request returns a result.
- *non-recursive server-controlled navigation* -- The client sends the request to a server and the server continues on behalf of the client, as above.
- *recursive server-controlled navigation* -- The client sends the request to a server and the server sends the request to another server (if needed) recursively.

Caching



Caching is a key technique to achieving performance for name services. The binding of names to attributes including other names is observed to not frequently change in many circumstances.

The results of a name resolution request can be cached by the client and by the servers.

Caching is used to eliminate high level name servers from the navigation path and allows resolution to proceed despite some server failures.

Of course, cached data may be out of date and changes can take time to propagate through the system.

Domain Name System

The Domain Name System (DNS) is a name service design whose main naming database is used across the Internet.

Before DNS, all host names and addresses were held in a single central master file and downloaded by FTP to all computers that required them.

The problems with the original name service included:

- It did not scale to large numbers of computers.
- Local organizations wished to administer their own naming systems.
- A general name service was needed -- not one that serves only for looking up computer addresses.

DNS is designed for use in multiple implementations, each of which may have its own name space, though in practice the Internet DNS name space is the one in widespread use.

DNS naming

The DNS name space is partitioned both organizationally and according to geography, though the domain name itself does not force the named resource to be located in any particular place or location.

DNS domain names are hierarchical from right to left, with "." characters used as a separator. E.g. the original highest level domains were `com`, `edu`, `gov`, `mil`, `net`, `org` and `int`. Country domains are `au`, `us`, `uk`, etc.

Each domain authority can specify their own subdomains, e.g. the UK uses `co.uk` and `ac.uk` for companies and academic communities (edu) respectively.

DNS queries

In general applications use the DNS to resolve host names in IP addresses. E.g. a query for DNS name `www.unimelb.edu.au`:

```
aharwood:~\$ host www.unimelb.edu.au
www.unimelb.edu.au has address 128.250.6.182
```

The UNIX `host` command can be used to make DNS queries. It formats the responses and prints them for the user to read. While an IPv4 address has exactly four parts to it, i.e. it is 32 bits longs, a domain name can have any number of subdomains.

DNS can also be used to make requests for certain services that support a domain, a common example being the mail host for a domain. E.g. the DNS request for the mail host for domain `unimelb.edu.au`.

```
aharwood:~\$ host -t MX unimelb.edu.au
unimelb.edu.au mail is handled by 10 antispam4.its.unimelb.edu.au.
unimelb.edu.au mail is handled by 50 muwayb.ucs.unimelb.edu.au.
unimelb.edu.au mail is handled by 10 antispam1.its.unimelb.edu.au.
unimelb.edu.au mail is handled by 10 antispam2.its.unimelb.edu.au.
unimelb.edu.au mail is handled by 10 antispam3.its.unimelb.edu.au.
```

Subsequent calls to find the `unimelb.edu.au` mailhost will usually return the hosts in a different order. This is to help load balance incoming email. The lower the priority number (`10` or `50` in the example), the higher the priority that the mail server has, i.e. high priority mail servers should be chosen by the mail client over low priority servers.

Reverse resolution allows an IP address to be resolved into a domain name. E.g.:

```
aharwood:~\$ host 128.250.6.182
182.6.250.128.in-addr.arpa domain name pointer www.unimelb.edu.au.
```

Host information allows information about a host to be obtained. This is usually not provided by the administrator because it constitutes a security problem. E.g.:

```
aharwood:~\$ host -t HINFO www.unimelb.edu.au
www.unimelb.edu.au host information "None" "None"
```

Well-known service allows information about services that are run by a computer to be returned.

DNS name servers

The DNS database is distributed across a logical network of servers.

The DNS naming data are divided into *zones*. A zone contains the following data:

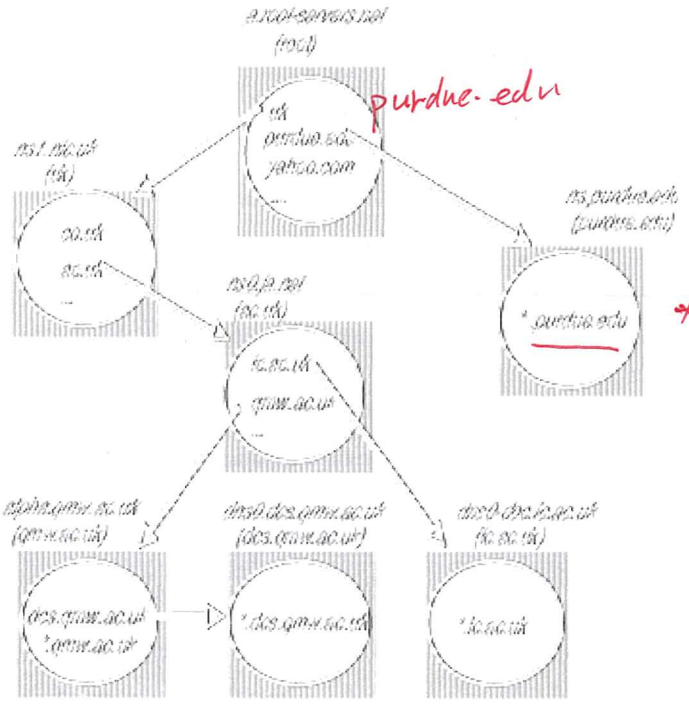
- Attribute data for names in a domain, less any sub-domains administered by lower-level authorities. E.g., a zone could contain data for `unimelb.edu.au` but not for `csse.unimelb.edu.au`.
- The names and addresses of at least two name servers that provide *authoritative* data for the zone.
- The names of the name servers that hold authoritative data for delegated sub-domains and the IP addresses of these servers.
- Zone management parameters, such as those governing the caching and replication of zone data.

The DNS architecture specifies that two name servers be provided for each domain, so that the name service can be available in the event of a single server crash.

A *primary* or *master server* reads zone data directly from a file. A *secondary server* downloads zone data from a primary server. Both of these kinds of servers are said to provide authoritative data for the zone. Secondary servers check and update their information on a regular basis, according to zone management parameters.

Any DNS server is free to cache data from other servers. A server that does so must let clients know that the data is not authoritative. Each entry in a zone has a time-to-live and a server must delete or update a cached entry after this time.

Example DNS domains



DNS resource records

Zone data are stored by name servers in files in one of several fixed types of resource record. For the Internet database these include:

- A -- A computer address, an IP number.
- NS -- A name server address, a domain name for the server.
- CNAME -- The canonical name for an alias, a domain name for alias.
- SOA -- Marks the start of data for a zone, parameters governing the zone.
- WKS -- A well-known service description, list of service names and protocols.
- PTR -- Domain name pointer (reverse lookups), a domain name.
- HINFO -- Host information, machine architecture and operating system.
- MX -- Mail exchange, list of (preference,host) pairs.
- TXT -- Text string, arbitrary text.

Example DNS zone file

```
\$TTL   86400 ; 24 hours could have been written as 24h or 1d
\$ORIGIN example.com.
@  1D  IN    SOA ns1.example.com.   hostmaster.example.com. (
                 2002022401 ; serial
                 3H ; refresh
                 15 ; retry
                 1w ; expire
                 3h ; minimum
                )
       IN  NS     ns1.example.com. ; in the domain
       IN  NS     ns2.smokeyjoe.com. ; external to domain
       IN  MX  10 mail.another.com. ; external mail provider
; server host definitions
ns1    IN  A      192.168.0.1  ;name server definition
www    IN  A      192.168.0.2  ;web server definition
ftp    IN  CNAME  www.example.com.  ;ftp server definition
; non server domain hosts
bill   IN  A      192.168.0.3
fred   IN  A      192.168.0.4
```

Distributed Systems
COMP90015 2019 SM1
File Systems
Lectures by Aaron Harwood
© University of Melbourne 2019

Summary

- file service architecture
- network file system
- enhancements and further developments

A basic *distributed file system* emulates the same functionality as a (non-distributed) file system for client programs running on multiple remote computers.

Advanced distributed file systems go much further by e.g. maintaining replica files and provide bandwidth and timing guarantees for multimedia data streaming.
A file system provides a convenient programming interface for disk storage along with features such as access control and file-locking that allows file sharing.

A *file service* allows programs to store and access remote files exactly as they do local ones, allowing users to access their files from any computer in an intranet.

Hosts that provide a file service can be optimized for persistent storage devices, e.g. for multiple disk drives, and can supply file services for a wide range of other services in an organization, e.g. for the web services and email services. This further facilitates management of the persistent storage, including backups and archiving.

Without using a distributed file system, a user must "manually" copy files from one machine to another, either using a network copy utility, by email, or by some other means like removable media.

Characteristics of file systems

Files contains both *data* and *attributes*. Data is usually in the form of a sequence of bytes and can be accessed and modified. Attributes include things like the length of the file, timestamps, file type, owner's identity and access-control lists.

Files have a name. Some files are directories that contain a list of other files; and they may themselves be (sub-) directories. This leads to a hierarchical naming scheme for the file. The *pathname* is the concatenation of the directory names and the file name.

Typical layers of a file system include:

- Directory module: relates file names to IDs
- File module: relates file IDs to particular files
- Access control module: checks permission for operation requested

- File access module: reads or writes file data or attributes
- Block module: access and allocates disk blocks
- Device module: disk I/O and buffering

UNIX file system operations are shown below:

- `fdes=open(name,mode)` - Opens an existing file with the given name
- `fdes=creat(name,mode)` - Creates a new file with the given name
- `status=close(fdes)` - Closes the open file
- `cnt=read(fdes,buf,n)` - Transfers bytes from the file into the buffer
- `cnt=write(fdes,buf,n)` - Transfers bytes from the buffer into the file
- `pos=lseek(fdes,offs,whence)` - Moves the read-write pointer to a new position in the file
- `status=unlink(name)` - Removes the file name from the directory structure
- `status=link(name1,name2)` - Adds a new name for the first named file
- `status=stat(name,buf)` - Gets the file attributes for the file

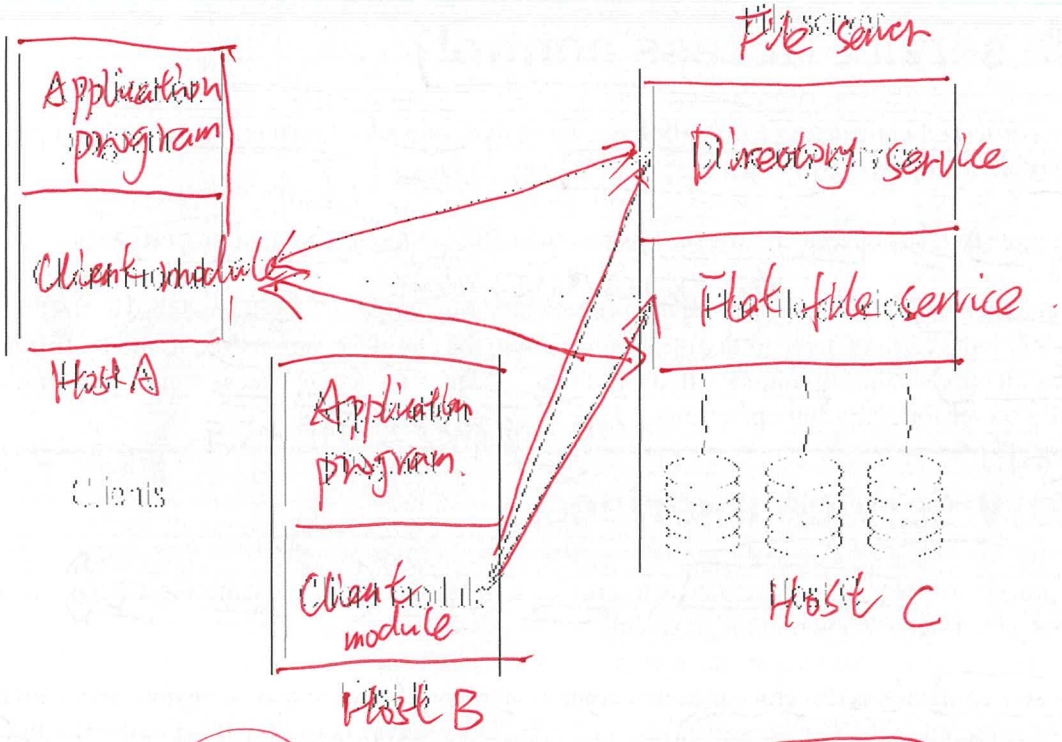
Distributed file system requirements

- **Transparency**:
  - *Access transparency* -- Client programs should be unaware of the distribution of files. Same API is used for accessing local and remote files and so programs written to operate on local files can, unchanged, operate on remote files.
  - *Location transparency* -- Client programs should see a uniform file name space; the names of files should be consistent regardless of where the files are actually stored and where the clients are accessing them from.
  - *Mobility transparency* -- Client programs and client administration services do not need to change when the files are moved from one place to another.
  - *Performance transparency* -- Client programs should continue to perform satisfactorily while the load on the service varies within a specified range.
  - *Scaling transparency* -- The service can be expanded by incremental growth to deal with a wide range of loads and network sizes.
- **Concurrent file updates**: Multiple clients' updates to files should not interfere with each other. Policies should be manageable.

- **File replication**: Each file can have multiple copies distributed over several servers, that provides better capacity for accessing the file and better fault tolerance.
- **Hardware and operating system heterogeneity**: The service should not require the client or server to have specific hardware or operating system dependencies.
- **Fault tolerance**: Transient communication problems should not lead to file corruption. Servers can use at-most-once invocation semantics or the simpler at-least-once semantics with *idempotent* operations. Servers can also be *stateless*.
- **Consistency**: Multiple, possibly concurrent, access to a file should see a consistent representation of that file, i.e. differences in the files location or update latencies should not lead to the file looking different at different times. File meta data should be consistently represented on all clients.
- **Security**: Client requests should be authenticated and data transfer should be encrypted.
- **Efficiency**: Should be of a comparable level of performance to conventional file systems.

File service architecture

- *Flat file service*: The flat file service is concerned with implementing operations on the contents of files. A *unique file identifier* (UFID) is given to the flat file service to refer to the file to be operated on. The UFID is unique over all the files in the distributed system. The flat file service creates a new UFID for each new file that it creates.
- *Directory service*: The directory service provides a mapping between text names and their UFIDs. The directory service creates directories and can add and delete files from the directories. The directory service is itself a client of the flat file service since the directory files are stored there.
- *Client module*: The client module integrates the directory service and flat file service to provide whatever application programming interface is expected by the application programs. The client module maintains a list of available file servers. It can also cache data in order to improve performance.



Though the client module is shown as directly supporting application programs, in practice it integrates into a virtual file system.

Flat file service interface

The flat file service interface is shown in the next slide, in terms of RPC calls.

Recall that the UNIX interface shown earlier requires that the UNIX file system maintains state, i.e. a file pointer, that is manipulated during reads and writes.

The flat file service interface differs from the UNIX interface mainly for reasons of fault tolerance:

- *repeatable operations* -- with the exception of `Create()`, the operations are idempotent, allowing the use of at-least-once RPC semantics.

- *stateless server* -- the flat file service does not need to maintain any state and can be restarted after a failure and resume operation without any need for clients or the server to restore any state.

Also note that UNIX files require an explicit open command before they can be accessed, while files in the flat file service can be accessed immediately.

- `Read(UFID, i, n) -> Data` - Reads up to n items from position i in the file
- `Write(UFID, i, Data)` - Writes the data starting at position i in the file. The file is extended if necessary
- `Create() -> UFID` - Creates a new file of length 0 and returns a UFID for it
- `Delete(UFID)` - Removes the file from the file store
- `GetAttributes(UFID) -> Attr` - Returns the file attributes for the file
- `SetAttributes(UFID,Attr)` - Sets the file attributes.

Flat file service access control

The service needs to authenticate the RPC caller and needs to ensure that illegal operations are not performed, e.g. that UFIDs are legal and that files access privileges are not ignored.

The server cannot store any access control state as this would break the idempotent property.

- An access check can be made whenever a file name is converted to a UFID and the results can be encoded in the form of a capability that is returned to the client for submission to the flat file server.
- A user identity can be submitted with every client request, and access checks can be performed by the flat file server for every file operation.

Directory service interface

The primary purpose of the directory service is to provide a translation from file names to UFIDs. An abstract directory service interface is shown in the next slide.

The directory server maintains directory files that contain mappings between text file names and UFIDs. The directory files are stored in the flat file server and so the directory server is itself a client to the flat file server.

A hierarchical file system can be built up from repeated accesses. E.g., the root directory has name "/" and the contains subdirectories with names "usr", "home", "etc", which themselves contain other subdirectories or files. A client function can make requests for the UFIDs in turn, to proceed through the path to the file or directory at the end.

- `Lookup(Dir, Name) -> UFID` - Returns the UFID for the file name in the given directory
- `AddName(Dir, Name, UFID)` - Adds the file name with UFID to the directory
- `UnName(Dir, Name)` - Remove the file name from the directory
- `GetNames(Dir, Pattern) -> Names` - Returns all the names in the directory that match the pattern.

File group

A *file group* is a collection of files located on a given server. A server may hold several file groups and file groups can be moved between servers, but a file cannot change file group.

File groups allow the file service to be implemented over several servers.

Files are given UFIDs that ensure uniqueness across different servers, e.g. by concatenating the server IP address (32 bits) with a date that the file was created (16 bits). This allows the files in a group, i.e. that have a common part to their UFID called the *file group identifier* to be relocated to a different server without conflicting with files already on that server.

The file service needs to maintain a mapping of UFIDs to servers. This can be cached at the client module.

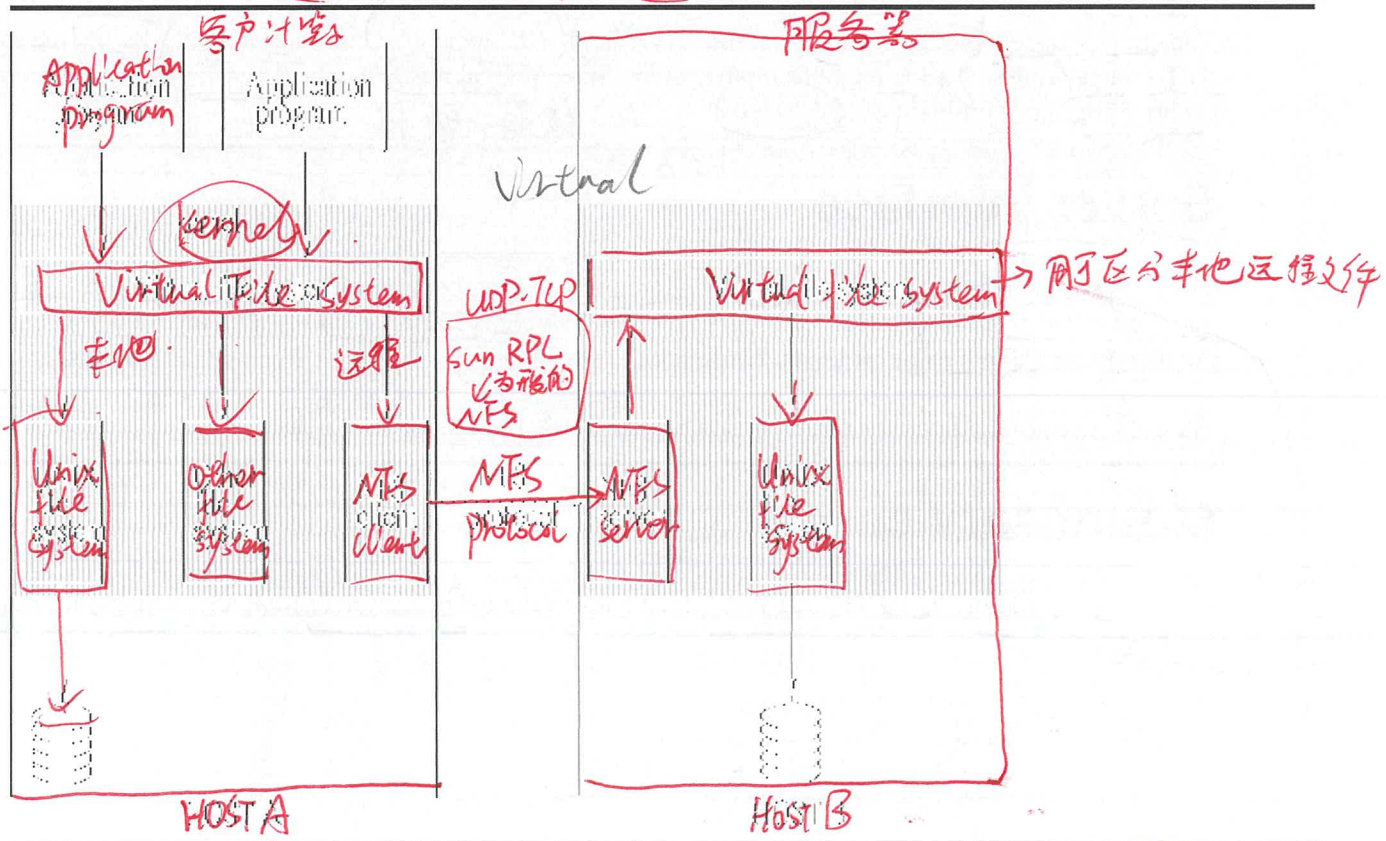
Sun Network File System

The Sun Network File System (NFS) follows the abstract system shown earlier.

There are many implementations of NFS and they all follow the NFS protocol using a set of RPCs that provide the means for the client to perform operations on the remote file store.

We consider a UNIX implementation.

The NFS client makes requests to the NFS server to access files.



Virtual file system

UNIX uses a *virtual file system* (VFS) to provide transparent access to any number of different file systems. The NFS is integrated in the same way.

The VFS maintains a VFS structure for each filesystem in use. The VFS structure relates a remote filesystem to the local filesystem; i.e. it combines the remote and local file system into a single filesystem.

The VFS maintains a *v-node* for each open file, and this records an indicator as to whether the file is local or remote.

If the file is local then the v-node contains a reference to the file's *i-node* on the UNIX file system.

If the file is remote then the v-node contains a reference to the file's NFS *file handle* which is a combination of *filesystem identifier*, i-node number and whatever else the NFS server needs to identify the file.

Client integration

The NFS client is integrated within the kernel so that:

- user programs can access files via UNIX system calls without recompilation or reloading;
- a single client module serves all of the user-level processes, with a shared cache;
- the encryption key used to authenticate user IDs passed to the server can be retained in the kernel, preventing impersonation by user-level clients.

The client transfers blocks of files from the server host to the local host and caches them, sharing the same buffer cache as used for local input-output system. Since several hosts may be accessing the same remote file, caching presents a problem of consistency.

Server interface

The NFS server interface integrates both the directory and file operations in a single service. The creation and insertion of file names in directories is performed by a single create operation, which takes the text name of the new file and file handle for the target directory as arguments.

The primitives of the interface largely resemble the UNIX filesystem primitives.

Mount service

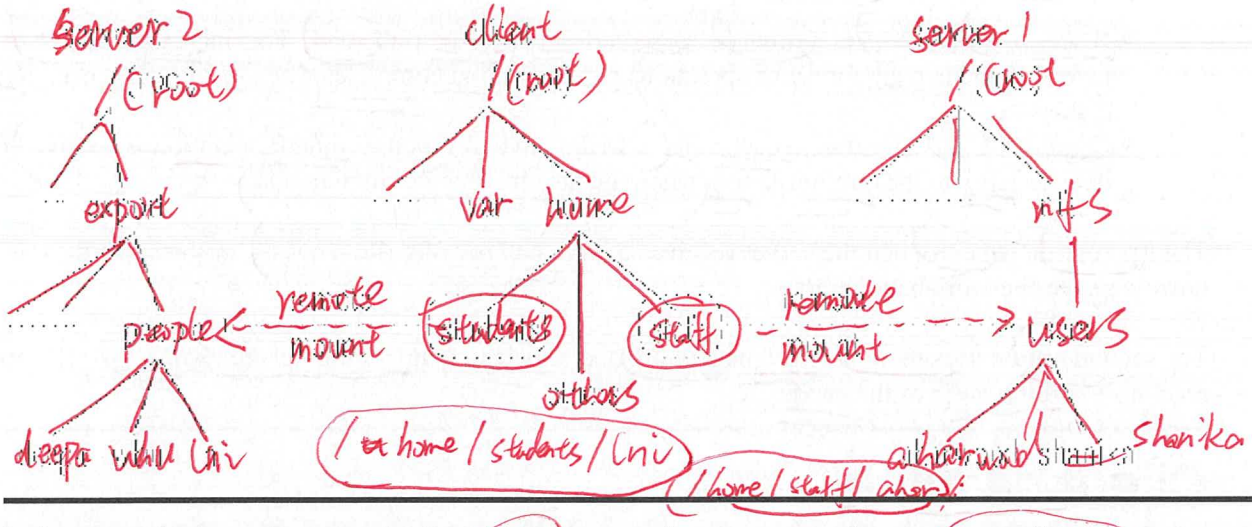
Each server maintains a file that describes which parts of the local filesystems that are available for remote mounting.

```
aharwood@htpc:~$ cat /etc/exports
# /etc/exports: the access control list for
#               filesystems which may be exported
#               to NFS clients.  See exports(5).
/store          192.168.1.0/255.255.255.0(rw)
```

In the above example all hosts on the subnet can mount the filesystem directory `/store` with read and write access.

A *hard-mounted* filesystem will block on each access until the access is complete. A *soft-mounted* filesystem will retry a few times and then return an error to the calling process.

Example NFS mounting



In the example `/etc/fstab`, the line starting `htpc:/store` states that the remote directory on `htpc` called `/store` should be mounted on the local filesystem; the name of the local mount point is given by the second parameter `/mnt/store`.

```
[aharwood@home ~]$ cat /etc/fstab
# This file is edited by fstab-sync - see 'man fstab-sync' for details
/dev/VolGroup00/LogVol00 /                       ext3    defaults        1 1
LABEL=/boot              /boot                   ext3    defaults        1 2
none                     /dev/pts                devpts  gid=5,mode=620  0 0
none                     /dev/shm                tmpfs   defaults        0 0
none                     /proc                   proc    defaults        0 0
none                     /sys                    sysfs   defaults        0 0
/dev/VolGroup00/LogVol01 swap                    swap    defaults        0 0
htpc:/store              /mnt/store              nfs     nfsvers=3,wsize=8192,rsize=8192,rw      0 0
/dev/hdc                 /media/cdrecorder       auto pamconsole,exec,noauto,managed 0 0
```

Server caching

In conventional UNIX systems, data read from the disk or pages are retained in a main memory buffer cache and are evicted when the buffer space is required for other pages. Accesses to cached data does not require a disk access.

Read-ahead anticipates read accesses and fetches the pages following those that have been recently read.

Delayed-write or write-back optimizes writes to the disk by only writing pages when both they have been modified and when they are evicted. A UNIX `sync` operation flushes modified pages to disk every 30 seconds.

This works for a conventional filesystem, on a single host, because there is only one cache and all file

accesses cannot bypass the cache.

Use of the cache at the server for client reads does not introduce any problems.

However use of the cache for writes requires special care to ensure that client can be confident that the writes are persistent, especially in the event of of a server crash.

There are two options for cache policies that are used by the server:

- *Write-through* -- data is written to cache and also directly to the disk. This increases disk I/O and increases the latency for write operations. The operation completes when the data has been written to disk.
- *Commit* -- data is written to cache and is written to disk when a commit operation is received for the data. A reply to the commit is sent when the data has been written to disk.

The first option is poor when the server receives a large number of write requests for the same data. It however saves network bandwidth.

The second option uses more network bandwidth and may lead to uncommitted data being lost. However it receives the full benefit of the cache.

Client caching

The NFS client also caches data reads, writes, attributes and directory operations in order to reduce network I/O.

Caching at the client introduces the cache consistency problem since now there is a cache at the client and the server, and there may be more than one client as well, each with its own cache.

Note that reading is a problem as well as writing, because a write on another client in between two read operations will lead to the second read operation being incorrect.

In NFS, clients poll the server to check for updates.

Let **T_c** be the time when a cache block was last validated by the client.

Let **T_m** be the time when a block was last modified.

A cache block is said to be valid at time **T** if (i) **T-Tc < *t*** where **t** is a given _freshness interval, or (ii) the value of **T_m** at the client matches the value at the server:

(T-Tc < *t*) *or* (*T*{m,client}=T_{m,server})

A small value for **t** leads to a close approximation of one-copy consistency, at the cost of greater network I/O.

In Sun Solaris clients **t** is set adaptively in the range 3 to 30 seconds depending on file update frequency. The range is 30 to 60 seconds for directories, since there is a lower risk of concurrent update.

The validity check is made on each access to cache block. If the first half of the check is true then the second half of the check need not be made. The first half of the check does not require network I/O.

A separate ***T{m,server}*** is kept by the server for the file attributes. If the first half of the check is found to false then the client contacts the server and retrieves **T{m,server}** for the file attributes. If it matches ***T{m,client}*** then the cache block is valid and the client sets **Tc** to the current time. If they do not match then the cache block is invalid and the client must request a new copy from the server.

Traffic can be reduced by applying new values of **T_{m,server}** to all relevant cache blocks and by piggy-backing attribute values on every file operation.

Write-back is used for writes, where modified files are flushed when a file is closed or when a sync operation takes place in the VFS. Special purpose daemons are used to do this asynchronously.

NFS summary

- *Access transparency*: Yes. Applications programs are usually not aware that files are remote and no changes are need to applications in order to access remote files.
- *Location transparency*: Not enforced. NFS does not enforce a global namespace since client filesystems may mount shared filesystems at different points. Thus an application that works on one client may not work on another.
- *Mobility transparency*: No. If the server changes then each client must be updated.
- *Scalability*: Good, could be better. The system can grow to accommodate more file servers as needed. Bottlenecks are seen when many processes access a single file.
- *File replication*: Not supported for updates. Additional services can be used to facilitate this.
- *Hardware and operating system heterogeneity*: Good. NFS is implemented on almost every known operating system and hardware platform.
- *Fault tolerance*: Acceptable. NFS is stateless and idempotent. Options exist for how to handle failures.

- *Consistency*: Tunable. NFS is not recommended for close synchronization between processes.
- *Security*: Kerberos is integrated with NFS. Secure RPC is also an option being developed.
- *Efficiency*: Acceptable. Many options exist for tuning NFS.