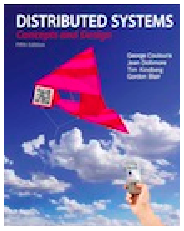

Distributed Systems: Concepts and Design

Edition 5

By George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair
Addison-Wesley ©Pearson Education 2012

Chapter 7 Exercise Solutions

7.1 Discuss each of the tasks of the core operating system's components – process management, thread management, communication management, memory management and supervisor – for UNIX (or any other OS you are familiar with).

7.1 Ans.

The core OS components are the following:

Process manager: Handles the creation of and operations upon processes. A process is a unit of resource management, including an address space and one or more threads.

Thread manager: Thread creation, synchronization and scheduling. Threads are schedulable activities attached to processes.

Communication manager: Communication between threads attached to different processes on the same computer. Some kernels also support communication between threads in remote processes. Other kernels have no notion of other computers built into them, and an additional service is required for external communication.

Memory manager: Management of physical and virtual memory. It is the utilization of memory management techniques for efficient data copying and sharing.

Supervisor: Dispatching of interrupts, system call traps and other exceptions; control of memory management unit and hardware caches; processor and floating point unit register manipulations. This is known as the Hardware Abstraction Layer in Windows.

7.2 How does the combination of middleware and network operating systems enable users to maintain a degree of autonomy for their machines?

7.2 Ans.

The combination of middleware and network operating systems provides an acceptable balance between the requirement for autonomy, on the one hand, and network-transparent resource access on the other. The network operating system enables users to run their favorite word processor and other standalone applications. Middleware enables them to take advantage of services that become available in their distributed system.

7.3 Smith decides that every thread in his processes ought to have its own *protected* stack – all other regions in a process would be fully shared. Does this make sense?

7.3 Ans.

If every thread has its own *protected* stack, then each must have its own address space. Smith's idea is better described as a set of single-threaded processes, most of whose regions are shared. The advantage of sharing an address space has thus been lost.

7.4 Do threats to a system's integrity come only from maliciously contrived code? Discuss.

7.4 Ans.

The threat to a system's integrity does not come only from maliciously contrived code. Benign code that contains a bug or which has unanticipated behaviour may cause part of the rest of the system to behave incorrectly.

7.5 Explain how a kernel provides protection to different processes in a system.

7.5 Ans.

A kernel process executes with the processor in supervisor (privileged) mode; the kernel arranges that other processes execute in user (unprivileged) mode.

The kernel also sets up address spaces to protect itself and other processes from the accesses of an aberrant process, and to provide processes with their required virtual memory layout. An address space is a collection of ranges of virtual memory locations, in each of which a specified combination of memory access rights applies, such as read-only or read-write. A process cannot access memory outside its address space. The terms user process or user-level process are normally used to describe one that executes in user mode and has a user-level address space (that is, one with restricted memory access rights compared with the kernel's address space).

7.6 Suggest a scheme for balancing the load on a set of computers. You should discuss:

i) what user or system requirements are met by such a scheme;

ii) to what categories of applications it is suited;

iii) how to measure load and with what accuracy; and

iv) how to monitor load and choose the location for a new process. Assume that processes may not be migrated.

How would your design be affected if processes could be migrated between computers? Would you expect process migration to have a significant cost?

7.6 Ans.

The following brief comments are given by way of suggestion, and are not intended to be comprehensive:

i) Examples of requirements, which an implementation might or might not be designed to meet: good interactive response times despite load level; rapid turnaround of individual compute-intensive jobs; simultaneous scheduling of a set of jobs belonging to a parallel application; limit on load difference between least- and most-loaded computers; jobs may be run on otherwise idle or under-utilized workstations; high throughput, in terms of number of jobs run per second; prioritization of jobs.

ii) A load-balancing scheme should be designed according to a job profile. For example, job behaviour (total execution time, resource requirements) might be known or unknown in advance; jobs might be typically interactive or typically compute-intensive or a mixture of the two; jobs may be parts of single parallel programs. Their total run-time may on average be one second or ten minutes. The efficacy of load balancing is doubtful for very light jobs; for short jobs, the system overheads for a complex algorithm may outweigh the advantages.

iii) A simple and effective way to measure a computer's load is to find the length of its run queue. Measuring load using a crude set of categories such as LIGHT and HEAVY is often sufficient, given the overheads of collecting finer-grained information, and the tendency for loads to change over short periods.

iv) A useful approach is for computers whose load is LIGHT to advertise this fact to others, so that new jobs are started on these computers.

Because of unpredictable job run times, any algorithm might lead to unbalanced computers, with a temporary dearth of new jobs to place on the LIGHT computers. If process migration is available, however, then jobs can be relocated from HEAVY computers to LIGHT computers at such times. The main cost of process migration is address space transfer, although techniques exist to minimise this [Kindberg 1990]. It can take in the order of seconds to migrate a process, and this time must be short in relation to the remaining run times of the processes concerned.

7.7 What are the advantages of using multi-threaded processes? Give an example of a situation where multi-threading reduces the tendency for servers to become bottlenecks.

7.7 Ans.

One of the advantages of multiple threads of execution is to maximize the degree of concurrent execution between operations, thus enabling the overlap of computation with input and output, and enabling concurrent processing on multiprocessors.

Multi-threading provides concurrent processing of clients' requests can reduce the tendency for servers to become bottlenecks. For example, one thread can process a client's request while a second thread servicing another request waits for a disk access to complete.

7.8 A file server uses caching, and achieves a hit rate of 80%. File operations in the server cost 5 ms of CPU time when the server finds the requested block in the cache, and take an additional 15 ms of disk I/O time otherwise. Explaining any assumptions you make, estimate the server's throughput capacity (average requests/sec) if it is:

i) single-threaded;

ii) two-threaded, running on a single processor;

iii) two-threaded, running on a two-processor computer.

7.8 Ans.

80% of accesses cost 5 ms; 20% of accesses cost 20 ms.

average request time is 0.8*5+.2*20 = 4+4=8ms.

i) single-threaded: rate is 1000/8 = 125 reqs/sec

ii) two-threaded: serving 4 cached and 1 uncached requests takes 25 ms. (overlap I/O with computation). Therefore throughput becomes 1 request in 5 ms. on average, = 200 reqs/sec

iii) two-threaded, 2 CPUs: Processors can serve 2 rqsts in 5 ms => 400 reqs/sec. But disk can serve the 20% of requests at only 1000/15 reqs/sec (assume disk rqsts serialised). This implies a total rate of 5*1000/15 = 333 requests/sec (which the two CPUs can service).

7.9 Threads share an execution environment. What are the different memory regions that different processes share?

7.9 Ans.

Different memory regions that different processes shares are:

a. Libraries

b. Kernel

c. Data sharing and communication.

7.10 What are the different policies for process allocation decisions?

7.10 Ans.

The process allocation decision – is a matter of policy. In general, process allocation policies range from always running new processes at their originator's workstation to sharing the processing load between a set of computers. These policies can be broadly divided into two parts –

a. The transfer policy determines whether to situate a new process locally or remotely. This may depend, for example, on whether the local node is lightly or heavily loaded.

b. The location policy determines which node should host a new process selected for transfer. This decision may depend on the relative loads of nodes, on their machine architectures and on any specialized resources they may possess.

7.11 A spin lock (see Bacon [2002]) is a boolean variable accessed via an atomic *test-and-set* instruction, which is used to obtain mutual exclusion. Would you use a spin lock to obtain mutual exclusion between threads on a single-processor computer?

7.11 Ans.

The problem that might arise is the situation in which a thread spinning on a lock uses up its timeslice, when meanwhile the thread that is about to free the lock lies idle on the READY queue. We can try to avoid this problem by integrating lock management with the scheduling mechanism, but it is doubtful whether this would have any advantages over a mutual exclusion mechanism without busy-waiting.

7.12 Explain thread-per-request and thread-per-connection architecture with regard to multi-threaded servers.

7.12 Ans.

In thread-per-request architecture the I/O thread spawns a new worker thread for each request, and that worker destroys itself when it has processed the request against its designated remote object. This architecture has the advantage that the threads do not contend for a shared queue, and throughput is potentially maximized because the I/O thread can create as many workers as there are outstanding requests. Its disadvantage is the overhead of the thread creation and destruction operations. (refer to Figure 7.6 a in the book.)

The thread-per-connection architecture associates a thread with each connection. The server creates a new worker thread when a client makes a connection and destroys the thread when the client closes the connection. In between, the client may make many requests over the connection, targeted at one or more remote objects. (refer to Figure 7.6 b in the book.)

7.13 Does switching present a problem for threads implementation?

7.13 Ans.

Switching between threads that is, running one thread instead of another at a given processor is advantageous. This cost is the most important factor, because it may be incurred many times in the lifetime of a thread. Switching between threads sharing the same execution environment is considerably cheaper than switching between different processes.

7.14 Explain the different methods available in Java to support synchronized execution among different threads in a system.

7.14 Ans.

The different methods to support synchronized execution in Java are

thread.join(long 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.

7.15 Why should a threads package be interested in the events of a thread's becoming blocked or unblocked? Why should it be interested in the event of a virtual processor's impending preemption? (Hint: other virtual processors may continue to be allocated.)

7.15 Ans.

If a thread becomes blocked, the user-level scheduler may have a READY thread to schedule. If a thread becomes unblocked, it may become the highest-priority thread and so should be run.

If a virtual processor is to be preempted, then the user-level scheduler may re-assign user-level threads to virtual processors, so that the highest-priority threads will continue to run.

7.16 Network transmission time accounts for 20% of a null RPC and 80% of an RPC that transmits 1024 user bytes (less than the size of a network packet). By what percentage will the times for these two operations improve if the network is upgraded from 10 megabits/second to 100 megabits/second?

7.16 Ans.

T_{null} = null RPC time = f + w_{null}, where f = fixed OS costs, w_{null} = time on wire at 10 megabits-per-second.

Similarly, T_{1024} = time for RPC transferring 1024 bytes = f + w_{1024}.

Let T'_{null} and T'_{1024} be the corresponding figures at 100 megabits per second. Then

$T'_{null} = f + 0.1w_{null}$, and $T'_{1024} = f + 0.1w_{1024}$.

Percentage change for the null RPC = $100(T_{null} - T'_{null})/T_{null} = 100*0.9w_{null} /T_{null} = 90*0.2 = 18\%$.

Similarly, percentage change for 1024-byte RPC = 100*0.9*0.8 = 72%.

7.17 A 'null' RMI that takes no parameters, calls an empty procedure and returns no values delays the caller for 2.0 milliseconds. Explain what contributes to this time.

In the same RMI system, each 1K of user data adds an extra 1.5 milliseconds. A client wishes to fetch 32K of data from a file server. Should it use one 32K RMI or 32 1K RMIs?

7.17 Ans.

Page 236 details the costs that make up the delay of a null RMI.

one 32K RMI: total delay is 2 + 32*1.5 = 50 ms.

32 1K RMIs: total delay is 32(2+1.5) = 112 ms -- one RMI is much cheaper.

7.18 Which kind of scheduling avoids race conditions? Which factors decide whether a thread should be scheduled as non-preemptive? *page 300*

7.18 Ans.

Non-preemptive scheduling avoids race conditions. Any thread which requires to be executed exclusively should be scheduled as non-preemptive.

7.19 Explain how it is possible to combine the advantages of user-level and kernel-level threads implementations.

7.19 Ans.

It is possible to combine the advantages of user-level and kernel-level threads implementations-

a. One approach, applied, for example, to the Mach kernel to enable user-level code to provide scheduling hints to the kernel's thread scheduler.

b. Another, adopted in the Solaris 2 operating system, is a form of hierarchical scheduling. Each process creates one or more kernel-level threads, known in Solaris as 'lightweight processes'. User-level threads are also supported. A user-level scheduler assigns each user-level thread to a kernel-level thread. This scheme can take advantage of multiprocessors, and also benefit because some thread-creation and thread-switching operations take place at user level.

7.20 i) Can a server invoked by lightweight procedure calls control the degree of concurrency within it?

ii) Explain why and how a client is prevented from calling arbitrary code within a server under lightweight RPC.

iii) Does LRPC expose clients and servers to greater risks of mutual interference than conventional RPC (given the sharing of memory)?

7.20 Ans.

i) Although a server using LRPC does not explicitly create and manage threads, it can control the degree of concurrency within it by using semaphores within the operations that it exports.

ii) A client must not be allowed to call arbitrary code within the server, since it could corrupt the server's data. The kernel ensures that only valid procedures are called when it mediates the thread's upcall into the server, as explained in Section 6.5.

iii) In principle, a client thread could modify a call's arguments on the A-stack, while another of the client's threads, executing within the server, reads these arguments. Threads within servers should therefore copy all arguments into a private region before attempting to validate and use them. Otherwise, a server's data is entirely protected by the LRPC invocation mechanism.

7.21 A client makes RMIs to a server. The client takes 5 ms to compute the arguments for each request, and the server takes 10ms to process each request. The local OS processing time for each *send* or *receive* operation is 0.5 ms, and the network time to transmit each request or reply message is 3 ms. Marshalling or unmarshalling takes 0.5 ms per message.

Estimate the time taken by the client to generate and return from 2 requests (i) if it is single-threaded, and (ii) if it has two threads which can make requests concurrently on a single processor. Is there a need for asynchronous RMI if processes are multi-threaded?

7.21 Ans.

(i) Single-threaded time: 2(5 (prepare) + 4(0.5 (marsh/unmarsh) + 0.5 (local OS)) + 2*3 (net)) + 10 (serv))
= 50 ms.

(ii) Two-threaded time: (see figure 6.14) because of the overlap, the total is that of the time for the first operation's request message to reach the server, for the server to perform all processing of both request and reply messages without interruption, and for the second operation's reply message to reach the client.
This is: 5 + (0.5+0.5+3) + (0.5+0.5+10+0.5+0.5) + (0.5+0.5+10+0.5+0.5) + (3 + 0.5+0.5)
= 37ms.

7.22 What is asynchronous invocation? What are the significant changes that persistent asynchronous invocation provides over a conventional invocation mechanism?

7.22 Ans.

An asynchronous invocation is one that is performed asynchronously with respect to the caller. That is, it is made with a non-blocking call, which returns as soon as the invocation request message has been created and is ready for dispatch.

A system for persistent asynchronous invocation tries indefinitely to perform the invocation, until it is known to have succeeded or failed, or until the application cancels the invocation instead of a conventional invocation mechanism (synchronous or asynchronous) which is designed to fail after a given number of timeouts have occurred. But these short-term timeouts are often not appropriate where disconnections or very high latencies occur.

7.23 Explain the key features of a monolithic kernel. How does it differ from the microkernel approach?

7.23 Ans.

A monolithic kernel performs all basic operating system functions and executes in the order of megabytes of code and data, and that it is undifferentiated: it is coded in a non-modular way. The result is that to a large extent it is intractable: altering any individual software component to adapt it to changing requirements is difficult.

Microkernel design provides only the most basic abstractions, principally address spaces, threads and local interprocess communication; all other system services are provided by servers that are dynamically loaded at precisely those computers in the distributed system that require them. Here clients access system services using the kernel's message-based invocation mechanisms.

7.24 How is virtualization relevant to the provision of cloud computing?

7.24 Ans.

Virtualization is highly relevant to the provision of cloud computing. Cloud computing adopts a model where storage, computation and higher-level objects built over them are offered as a service. This ranges from low-level aspects such as physical infrastructure (referred to as infrastructure as a service), through software platforms such as the Google App Engine to arbitrary application-level services (software as a service). The first issue is enabled directly by virtualization, allowing users of the cloud to be provided with one or more virtual machines for their own use.

7.25 On a certain computer we estimate that, regardless of the OS it runs, thread scheduling costs about 50 µs, a null procedure call 1 µs, a context switch to the kernel 20 µs and a domain transition 40 µs. For each of Mach and SPIN, estimate the cost to a client of calling a dynamically loaded null procedure.

7.25 Ans.

Mach, by default, runs dynamically loaded code in a separate address space. So invoking the code involves control transfer to a thread in a separate address space. This involves four (context switch + domain transitions) to and from the kernel as well as two schedulings (client to server thread and server thread to client thread) -- in addition to the null procedure itself.

Estimated cost: 4(20 + 40) + 2*50 + 1 = 341 µs

In SPIN, the call involve two (context switch + domain transitions) and no thread scheduling.

Estimated cost: 2(20 + 40) + 1 = 121 µs.

7.26 What is the distinction between the virtualization approach advocated by Xen and the style of microkernel advocated by the Exokernel project? In your answer, highlight two things they have in common and two distinguishing characteristics between the approaches.

7.26 Ans.

Both approaches are examples of virtualization in the general sense of the word, offering abstraction over the underlying hardware.

They also both support the coexistence of a number of virtual machines on one physical architecture (and also the necessary isolation between instances), through domains in the case of virtualization and operating system emulations in the case of microkernels.

They do operate at different levels though. System virtualization is concerned with providing virtualization over the hardware, preserving the interface as provided by the hardware. In contrast, microkernels provide a much higher level of abstraction based on offering (typically policy neutral) operating system services (for example address spaces, threads and interprocess communication on a single machine).

The emphasis in system virtualization is on extremely lightweight implementation, hence the interest in techniques such as paravirtualization, and hence system virtualization can support potentially very large numbers of virtual machines compared to the microkernel approach.

The use cases of system virtualization are also quite different, for example managing the mapping of services to virtual machines in server farms (exploiting the ability to migrate virtual images).

7.27 Sketch out in pseudo-code how you would add a simple round robin scheduler to the Xen hypervisor using the framework discussed in Section 7.7.2.

7.27 Ans.

In order to answer this question, it is first necessary to appreciate the steps required to implement a new scheduler (as discussed in Chisnall [2007]).

To introduce anew scheduler, the following steps must be carried out:

- Provide an implementation of various functions defined in an abstract interface, with these functions including operations to initialise the scheduler, initialise and destroy a domain, initialise and destroy a VCPU, sleep, wake and a call to determine the next VCPU to be scheduled (*do_schedule*);

- Create a data structure (*struct Scheduler*) that contains the full name, short name and ID for this scheduler together with pointers to the above functions (for some schedulers certain functions can be NULL);
- Add this to a static array of schedulers with the user able to specify the scheduler to be used at boot time.

As an example, the following is a minimal specification of a simple scheduler, where *simple_vcpu_init*, *simple_vcpu_destroy* and *simple_schedule* are implementations of the respective functions:

```
struct scheduler sched_simpleRR_def = {
    .name          = "Simple Round Robin Scheduler",
    .opt_name      = "simpleRR",
    .sched_id      = 7,
    .init_vcpu     = simple_vcpu_init,
    .destroy_vcpu  = simple_vcpu_destroy,
    .do_schedule   = simple_schedule,
};
```

It is then necessary to implement the three functions above. Let us first assume the existence of a singly linked list of VCPUs. The pseudo code for the three functions is then as follows

```
simple_vcpu_init (struct vcpu *v)
{
    <insert the new VCPU, v,  into the linked list>
}
simple_vcpu_destroy (struct vcpu *v)
{
    <remove the VCPU, v, from the linked list>
}
simple_do_schedule ()
{
        <traverse the linked list and select the first runnable VCPU>
        <move the head to the tail>
};
```

7.28 From your understanding of the Xen approach to virtualization, discuss specific features of Xen that can support the XenoServer architecture, thus illustrating the synergy between virtualization and cloud computing.

7.28 Ans.

As explained in Section 7.7.2, the XenoServer architecture supports the discovery and use of XenoServers or indeed more sophisticated servers or cluster of servers with a given level of connectivity. The architecture therefore promotes a view of everything as a service as in cloud computing. Through using Xen, the mapping of XenoServers to physical hardware can be managed. More generally, the use of virtualization in cloud computing supports the provision of infrastructure as a service by providing access to virtual rather than physical machines.