



Distributed Systems: Concepts and Design

Edition 3

By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001

Chapter 6 Exercise Solutions

- 6.1 Discuss each of the tasks of encapsulation, concurrent processing, protection, name resolution, communication of parameters and results, and scheduling in the case of the UNIX file service (or that of another kernel that is familiar to you).

6.1 Ans.

We discuss the case of a single computer running Unix.

Encapsulation: a process may only access file data and attributes through the system call interface.

Concurrent processing: several processes may access the same or different files concurrently; a process that has made a system call executes in supervisor mode in the kernel; processes share all file-system-related data, including the block cache.

Protection: users set access permissions using the familiar *user/group/other, rwx* format. Address space protection and processor privilege settings are used to restrict access to file data and file system data in memory and prevent direct access to storage devices. Processes bear user and group identifiers in protected kernel tables, so the problem of authentication does not arise.

Name resolution: pathnames (for example, */usr/fred*) are resolved by looking up each component in turn. Each component is looked up in a directory, which is a table containing path name components and the corresponding inodes. If the inode is that of a directory then this is retrieved; and the process continues until the final component has been resolved or an error has occurred. Special cases occur when a symbolic link or mount point is encountered.

Parameter and result communication: parameters and results can be communicated by a) passing them in machine registers, b) copying them between the user and kernel address spaces, or c) by mapping data blocks simultaneously in the two address spaces.

Scheduling: there are no separate file system threads; when a user process makes a file system call, it continues execution in the kernel.

NFS is discussed in Section 8.2.

- 6.2 Why are some system interfaces implemented by dedicated system calls (to the kernel), and others on top of message-based system calls?

6.2 Ans.

Dedicated system calls are more efficient for simple calls than message-based calls (in which a system action is initiated by sending a message to the kernel, involving message construction, dispatch etc.).

However, the advantage of implementing a system call as an RPC is that then a process can perform operations transparently on either remote or local resources.

- 6.3 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?

6.3 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.

6.4 Should signal (software interrupt) handlers belong to a process or to a thread?

6.4 Ans.

When a process experiences a signal, should any currently running thread handle it, or should a pre-arranged thread handle it? And do threads have their own tables of signal handlers? There is no 'correct' answer to these questions. For example, it might be convenient for threads within a process to use their own SIGALRM signal handlers. On the other hand, a process only needs one SIGINT handler.

6.5 Discuss the issue of naming applied to shared memory regions.

6.5 Ans.

Memory regions may be given textual names. Two or more processes may then share a region by supplying its name and requesting it to be mapped into their address spaces. Mapped files are examples of this idea, in which the contents of the shared region are maintained on persistent store.

The second naming issue is that of the addresses used to access a shared region. In principle, a shared region may be mapped as different address ranges in different processes. However, it is more convenient if the same address ranges are used, for the region may then contain pointers into itself, without the need for translation. Unfortunately, in the absence of global address space management, some addresses belonging to a particular region might already be occupied in some processes.

6.6 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?

6.6 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.

6.7 Explain the advantage of copy-on-write region copying for UNIX, where a call to *fork* is typically followed by a call to *exec*. What should happen if a region that has been copied using copy-on-write is itself copied?

6.7 Ans.

It would be wasteful to copy the forked process's address space contents, since they are immediately replaced. With copy-on-write, only those few pages that are actually needed before calling *exec* will be copied.

Assume now that *exec* is not called. When a process forks, its child may in turn fork a child. Call these the parent, child and grandchild. Pages in the grandchild are logically copied from the child, whose pages are in turn logically copied from the parent. Initially, the grandchild's pages may share a frame with the parent's page. If, say, the child modifies a page, however, then the grandchild's page must be made to share the child's frame. A way must be found to manage chains of page dependencies, which have to be altered when pages are modified.

- 6.8 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:
- single-threaded;
 - two-threaded, running on a single processor;
 - two-threaded, running on a two-processor computer.

6.8 Ans.

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

average request time is $0.8*5 + .2*20 = 4+4=8\text{ms}$.

i) single-threaded: rate is $1000/8 = 125 \text{ reqs/sec}$

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 reqs in 5 ms => 400 reqs/sec. But disk can serve the 20% of requests at only 1000/15 reqs/sec (assume disk reqs serialised). This implies a total rate of $5*1000/15 = 333 \text{ requests/sec}$ (which the two CPUs can service).

- 6.9 Compare the worker pool multi-threading architecture with the thread-per-request architecture.

6.9 Ans.

The worker pool architecture saves on thread creation and destruction costs compared to the thread-per-request architecture but (a) the pool may contain too few threads to maximise performance under high workloads or too many threads for practical purposes and (b) threads contend for the shared work queue.

- 6.10 What thread operations are the most significant in cost?

6.10 Ans.

Thread switching tends to occur many times in the lifetime of threads and is therefore the most significant cost. Next come thread creation/destruction operations, which occur often in dynamic threading architectures (such as the thread-per-request architecture).

- 6.11 A spin lock (see Bacon [1998]) 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?

6.11 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.

- 6.12 Explain what the kernel must provide for a user-level implementation of threads, such as Java on UNIX.

6.12 Ans.

A thread that makes a blocking system call provides no opportunity for the user-level scheduler to intervene, and so all threads become blocked even though some may be in the READY state. A user-level implementation requires (a) non-blocking (asynchronous) I/O operations provided by the kernel, that only initiate I/O; and (b) a way of determining when the I/O has completed -- for example, the UNIX *select* system call. The threads programmer should not use native blocking system calls but calls in the threading API which make an asynchronous call and then invoke the scheduler.

6.13 Do page faults present a problem for user-level threads implementations?

6.13 Ans.

If a process with a user-level threads implementation takes a page fault, then by default the kernel will deschedule the entire process. In principle, the kernel could instead generate a software interrupt in the process, notifying it of the page fault and allowing it to schedule another thread while the page is fetched.

6.14 Explain the factors that motivate the hybrid scheduling approach of the 'scheduler activations' design (instead of pure user-level or kernel-level scheduling).

6.14 Ans.

A hybrid scheduling scheme combines the advantages of user-level scheduling with the degree of control of allocation of processors that comes from kernel-level implementations. Efficient, custom scheduling takes place inside processes, but the allocation of a multiprocessor's processors to processes can be globally controlled.

6.15 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.)

6.15 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.

6.16 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?

6.16 Ans.

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

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

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

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

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

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

6.17 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?

6.17 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.

6.18 Which factors identified in the cost of a remote invocation also feature in message passing?

6.18 Ans.

Most remote invocation costs also feature in message passing. However, if a sender uses asynchronous message passing then it is not delayed by scheduling, data copying at the receiver or waiting for acknowledgements

6.19 Explain how a shared region could be used for a process to read data written by the kernel. Include in your explanation what would be necessary for synchronization.

6.19 Ans.

The shared region is mapped read-only into the process's address space, but is writable by the kernel. The process reads data from the region using standard LOAD instructions (avoiding a TRAP). The process may poll the data in the region from time to time to see if it has changed. However, we may require a way for the kernel to notify the process when it has written new data. A software interrupt can be used for this purpose.

- 6.20 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)?

6.20 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.

- 6.21 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?

6.21 Ans.

(i) Single-threaded time: $2(5 \text{ (prepare)} + 4(0.5 \text{ (marsh/unmarsh)} + 0.5 \text{ (local OS)}) + 2*3 \text{ (net)}) + 10 \text{ (serv)}$
= 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.

- 6.22 Explain what is security policy and what are the corresponding mechanisms in the case of a multi-user operating system such as UNIX.

6.22 Ans.

Mechanisms: see answer to 6.1.

Policy concerns the application of these mechanisms by a particular user or in a particular working environment. For example, the default access permissions on new files might be "rw-----" in the case of an environment in which security is a high priority, and "rw-r--r--" where sharing is encouraged.

- 6.23 Explain the program linkage requirements that must be met if a server is to be dynamically loaded into the kernel's address space, and how these differ from the case of executing a server at user level.

6.23 Ans.

Portions of the kernel's address space must be allocated for the new code and data. Symbols within the new code and data must be resolved to items in the kernel's address space. For example, it would use the kernel's message-handling functions.

By contrast, if the server was to execute as a separate process then it would run from a standard address in its own address space and, apart from references to shared libraries, its linked image would be self-contained.

- 6.24 How could an interrupt be communicated to a user-level server?

6.24 Ans.

The interrupt handler creates a message and sends it, using a special non-blocking primitive, to a predetermined port which the user-level server owns.

- 6.25 On a certain computer we estimate that, regardless of the OS it runs, thread scheduling costs about $50 \mu\text{s}$, a null procedure call $1 \mu\text{s}$, a context switch to the kernel $20 \mu\text{s}$ and a domain transition $40 \mu\text{s}$. For each of Mach and SPIN, estimate the cost to a client of calling a dynamically loaded null procedure.

6.25 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: $4(20 + 40) + 2*50 + 1 = 341 \mu\text{s}$

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

Estimated cost: $2(20 + 40) + 1 = 121 \mu\text{s}$.