Chapter 6 Selected 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.

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.

6.8 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.

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=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).

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_{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%.

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.

6.21 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 (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.

Chapter 10 Selected Solutions

10.1 Why is computer clock synchronization necessary? Describe the design requirements for a system to synchronize the clocks in a distributed system.

10.1 Ans.

See Section 10.1 for the necessity for clock synchronization. Major design requirements:

i) there should be a limit on deviation between clocks or between any clock and UTC;

ii) clocks should only ever advance;

iii) only authorized principals may reset clocks (i.e. on Kerberos 7.6.2)

In practice (i) cannot be achieved unless only benign failures are assumed to occur and the system is synchronous.

10.3 A scheme for implementing at-most-once reliable message delivery uses synchronized clocks to reject duplicate messages. Processes place their local clock value (a 'timestamp') in the messages they send. Each receiver keeps a table giving, for each sending process, the largest message timestamp it has seen. Assume that clocks are synchronized to within 100 ms, and that messages can arrive at most 50 ms after transmission.

(i) When may a process ignore a message bearing a timestamp T, if it has recorded the last message received from that process as having timestamp T'?

(ii) When may a receiver remove a timestamp 175,000 (ms) from its table? (Hint: use the receiver's local clock value.)

(iii) Should the clocks be internally synchronized or externally synchronized?

10.3 Ans.

i) If $T <= T_{\ni}$ then the message must be a repeat.

ii) The earliest message timestamp that could still arrive when the receiver's clock is r is r - 100 - 50. If this is to be at least 175,000 (so that we cannot mistakenly receive a duplicate), we need r -150 = 175,000, i.e. r = 175,150.

iii) Internal synchronisation will suffice, since only time *differences* are relevant.

10.4 A client attempts to synchronize with a time server. It records the round-trip times and timestamps returned by the server in the table below.

Which of these times should it use to set its clock? To what time should it set it? Estimate the accuracy of the setting with respect to the server's clock. If it is known that the time between sending and receiving a message in the system concerned is at least 8 ms, do your answers change?

Round-trip (ms)	*Time (hr:min:sec)*
22	*10:54:23.674*
25	*10:54:25.450*
20	*10:54:28.342*

10.4 Ans.

The client should choose the minimum round-trip time of 20 ms = 0.02 s. It then estimates the current time to be 10:54:28.342 + 0.02/2 = 10:54:28.352. The accuracy is ± 10 ms. If the minimum message transfer time is known to be 8 ms, then the setting remains the same but the accuracy improves to ± 2 ms.

10.5 In the system of Exercise 10.4 it is required to synchronize a file server's clock to within ±1 millisecond. Discuss this in relation to Cristian's algorithm.

10.5 Ans.

To synchronize a clock within ± 1 ms it is necessary to obtain a round-trip time of no more than 18 ms, given the minimum message transmission time of 8 ms. In principle it is of course possible to obtain such a round-trip time, but it may be improbable that such a time could be found. The file server risks failing to synchronize over a long period, when it could synchronize with a lower accuracy.

10.6 What reconfigurations would you expect to occur in the NTP synchronization subnet?

10.6 Ans.

A server may fail or become unreachable. Servers that synchronize with it will then attempt to synchronize to a different server. As a result, they may move to a different stratum. For example, a stratum 2 peer (server) loses its connection to a stratum 1 peer, and must thenceforth use a stratum 2 peer that has retained its connection to a stratum 1 peer. It becomes a stratum 3 peer. Also, if a primary server's UTC source fails, then it becomes a secondary server.

10.7 An NTP server B receives server A's message at 16:34:23.480 bearing a timestamp 16:34:13.430 and replies to it. A receives the message at 16:34:15.725, bearing B's timestamp 16:34:25.7. Estimate the offset between B and A and the accuracy of the estimate.

10.7 Ans.

Let $a = T_{i-2} - T_{i-3}$ = 23.48 - 13.43 = 10.05; $b = T_{i-1} - T_i$ = 25.7- 15.725 =9.975.
Then the estimated offset $o_i = (a+b)/2$ = 10.013s, with estimated accuracy = $\pm d_i/2 = \pm (a-b)/2$ = 0.038s (answers expressed to the nearest millisecond).

10.10 By considering a chain of zero or more messages connecting events *e* and and using induction, show that e → e' ➔ L(e) < L(e')

.

10.10 Ans.

If *e* and are successive events occurring at the same process, or if there is a message *m* such that *e* = *send*(m) and = *rcv*(m), then the result is immediate from LC1 and LC2. Assume that the result to be proved is true for all pairs of events connected in a sequence of events (in which either *HB1* or *HB2* applies between each neighbouring pair) of length *N* or less ($N >= 2$). Now assume that *e* and e' are connected in a series of events e_1 , e_2 , e_3 , .., e_{N+1} occurring at one or more processes such that $e = e_1$ and e' = e_{N+1} . Then $e \longrightarrow e_N$ and so C(e) < C(e_N) by the induction hypothesis. But by LC1 and LC2, C(e_N) <C(e'). Therefore C(e) < C(e').

10.11 Show that (NB erratum in this exercise in first printing)

10.11 Ans.

Rule VC2 (p. 399) tells us that is the 'source' of increments to Vi[i] , which it makes just before it sends each message; and that pj increments Vj[i] only as it receives messages containing timestamps with larger entries for pi. The relationship Vj[i] <= Vi[i] follows immediately.

10.12 In a similar fashion to Exercise 10.10, show that e → e' → V(e) < V(e') .

10.12 Ans.
If *e* and e' are successive events occurring at the same process, or if there is a message *m* such that *e* = *sen*d(m) and e' = *rc*v(m), then the result follows from VC2–VC4. In the latter case the sender includes its timestamp value and the recipient increases its own vector clock entry; all of its other entries remain at least as great as those in the sender's timestamp. Assume that the result to be proved is true for all pairs of events connected in a sequence of events (in which either *HB1* or *HB2* applies between each neighbouring pair) of length *N* or less *(N* >=2). Now assume that *e* and e' are connected in a series of events e_1 , e_2 , e_3 , .., e_{N+1} occurring at one or more processes such that $e = e_1$ and e'= e_{N+1} . Then e → e_N and so V(e) < V(e_N) by the induction hypothesis. But by VC2–VC4, V(e_N) < V(e'). Therefore V(e) < V(e').

10.13 Using the result of Exercise 10.11, show that if events *e* and e' are concurrent then neither V(e) <= V(e') nor V(e') <= V(e). Hence show that if V(e) < V(e') then e → e'.

10.13 Ans.
Let *e* and e' be concurrent and let *e* occur at p_i and e' at p_j . Because the events are concurrent (not related by happened-before) we know that no message sent from p_i at or after event *e* has propagated its timestamp to p_j by the time e' occurs at p_j, and *vice vers*a. By the reasoning for Exercise 10.11, it follows that $V_j[\iota] < V_i[\iota]$ and Vi[j] < Vj[j] (strict inequalities) and therefore that neither V(e) <= V(e') nor V(e') <= V(e).

Therefore if V(e) < V(e') the two events are not concurrent – they must be related by happened-before. Of the two possibilities, it obviously must be that .e → e'.

10.14 Two processes *P* and *Q* are connected in a ring using two channels, and they constantly rotate a message m. At any one time, there is only one copy of *m* in the system. Each process's state consists of the number of times it has received m, and *P* sends *m* first. At a certain point, *P* has the message and its state is 101. Immediately after sending m, *P* initiates the snapshot algorithm. Explain the operation of the algorithm in this case, giving the possible global state(s) reported by it.

10.14 Ans.
P sends msg m
P records state (101)
P sends marker (see initiation of algorithm described on p. 406)
Q receives m, making its state 102
Q receives the marker and by marker-receiving rule, records its state (102) and the state of the channel from P to Q as {}
Q sends marker (marker-sending rule)
(Q sends m again at some point later)
P receives marker
P records the state of the channel from Q to P as set of messages received since it saved its state = {} (marker-receiving rule).

10.16 Jones is running a collection of processes p_1 p_2 ...p_N . Each process p_i contains a variable v_i. She wishes to determine whether all the variables v_1 v_2 … v_N were ever equal in the course of the execution.
(i) Jones' processes run in a synchronous system. She uses a monitor process to determine whether the variables were ever equal. When should the application processes communicate with the monitor process, and what should their messages contain?
(ii) Explain the statement *possibly (*v1 = v2 = … vN). How can Jones determine whether this statement is true of her execution?

10.16 Ans.
(i) communicate new value when local variable v_i changes;
with this value send: current time of day *C(e)* and vector timestamp *V(e)* of the event of the change, e.
(ii) *possibly* (...): there is a consistent, potentially simultaneous global state in which the given predicate is true.
Monitor process takes potentially simultaneous events which correspond to a consistent state, and checks predicate v1 = v2 = … vN.
Simultaneous: estimate simultaneity using bound on clock synchronization and upper limit on message propagation time, comparing values of *C* (see p. 415).
Consistent state: check vector timestamps of all pairs e_i , e_j of potentially simultaneous events , : check V(ei)[i] >= V(ej)[j].

Chapter 11 Selected Solutions

11.1 Is it possible to implement either a reliable or an unreliable (process) failure detector using an unreliable communication channel?
11.1 Ans.
An unreliable failure detector can be constructed from an unreliable channel – all that changes from use of a reliable channel is that dropped messages may increase the number of false suspicions of process failure. A reliable failure detector requires a synchronous system. It cannot be built on an unreliable channel since a dropped message and a failed process cannot be distinguished – unless the unreliability of the channel can be masked while providing a guaranteed upper bound on message delivery times. A channel that dropped messages with some probability but, say, guaranteed that at least one message in a hundred was not dropped could, in principle, be used to create a reliable failure detector.

11.4 In the central server algorithm for mutual exclusion, describe a situation in which two requests are not processed in happened-before order.
11.4 Ans.
Process *A* sends a request Ra for entry then sends a message *m* to B. On receipt of m, *B* sends request Rb for entry. To satisfy happened-before order, Ra should be granted before Rb. However, due to the vagaries of message propagation delay, Rb arrives at the server before Ra, and they are serviced in the opposite order.

11.5 Adapt the central server algorithm for mutual exclusion to handle the crash failure of any client (in any state), assuming that the server is correct and given a reliable failure detector. Comment on whether the

resultant system is fault tolerant. What would happen if a client that possesses the token is wrongly suspected to have failed?

11.5 Ans.

The server uses the reliable failure detector to determine whether any client has crashed. If the client has been granted the token then the server acts as if the client had returned the token. In case it subsequently receives the token from the client (which may have sent it before crashing), it ignores it. The resultant system is not fault-tolerant. If a token-holding client crashed then the application-specific data protected by the critical section (whose consistency is at stake) may be in an unknown state at the point when another client starts to access it. If a client that possesses the token is wrongly suspected to have failed then there is a danger that two processes will be allowed to execute in the critical section concurrently.

11.7 In a certain system, each process typically uses a critical section many times before another process requires it. Explain why Ricart and Agrawala's multicast-based mutual exclusion algorithm is inefficient for this case, and describe how to improve its performance. Does your adaptation satisfy liveness condition ME2?

11.7 Ans.

In Ricart and Agrawala's multicast-based mutual exclusion algorithm, a client issues a multicast request every time it requires entry. This is inefficient in the case described, of a client that repeatedly enters the critical section before another needs entry.

Instead, a client that finishes with a critical section and which has received no outstanding requests could mark the token as *JUST_RELEASE*D, meaning that it has not conveyed any information to other processes that it has finished with the critical section. If the client attempts to enter the critical section and finds the token to be *JUST_RELEASE*D, it can change the state to *HELD* and re-enter the critical section.

To meet liveness condition ME2, a *JUST_RELEASED* token should become *RELEASED* if a request for entry is received.

11.8 In the Bully algorithm, a recovering process starts an election and will become the new coordinator if it has a higher identifier than the current incumbent. Is this a necessary feature of the algorithm?

11.8 Ans.

First note that this is an undesirable feature if there is no advantage to using a higher-numbered process: the re-election is wasteful. However, the numbering of processes may reflect their relative advantage (for example, with higher-numbered processes executing at faster machines). In this case, the advantage may be worth the re-election costs. Re-election costs include the message rounds needed to implement the election; they also may include application-specific state transfer from the old coordinator to the new coordinator.

To avoid a re-election, a recovering process could merely send a *requestStatus* message to successive lower-numbered processes to discover whether another process is already elected, and elect itself only if it receives a negative response. Thereafter, the algorithm can operate as before: if the newly-recovered process discovers the coordinator to have failed, or if it receives an *election* message, it sends a *coordinator* message to the remaining processes.

11.9 Suggest how to adapt the Bully algorithm to deal with temporary network partitions (slow communication) and slow processes.

11.9 Ans.

With the operating assumptions stated in the question, we cannot guarantee to elect a unique process at any time. Instead, we may find it satisfactory to form subgroups of processes that agree on their coordinator, and allow several such subgroups to exist at one time. For example, if a network splits into two then we could form two subgroups, each of which elects the process with the highest identifier among its membership. However, if the partition should heal then the two groups should merge back into a single group with a single coordinator. The algorithm known as the 'invitation algorithm' achieves this. It elects a single coordinator among each subgroup whose members can communicate, but periodically a coordinator polls other members of the entire set of processes in an attempt to merge with other groups. When another coordinator is found, a coordinator sends it an 'invitation' message to invite it to form a merged group. As

in the Bully algorithm, when a process suspects the unreachability or failure of its coordinator it calls an election.

Chapter 12 Selected Solutions

12.2 A server manages the objects a1, a2,... *a*n. The server provides two operations for its clients:
read (i) returns the value of *a*i; *write(i, Value)* assigns *Value* to *a*i.
The transactions *T* and *U* are defined as follows:
T: x= read (j); y = read (i); write(j, 44); write(i, 33);
U: x= read(k); write(i, 55); y = read (j); write(k, 66).
Give two serially equivalent interleavings of the transactions *T* and U.

12.2 Ans.
The interleavings of *T* and *U* are serially equivalent if they produce the same outputs (in x and y) and have the same effect on the objects as some serial execution of the two transactions. The two possible serial executions and their effects are:
If *T* runs before U: $x_T = a_{j0}$; $y_T = a_{i0}$; $x_U = a_{k0}$; $y_U = 44$; $a_i = 55$; $a_j = 44$; $a_k = 66$.
If *U* runs before T: $x_T = a_{j0}$; $y_T = 55$; $x_U = a_{k0}$; $y_U = a_{j0}$; $a_i = 33$; $a_j = 44$; $a_k = 66$,
where x_T and y_T are the values of x and y in transaction T; x_U and y_U are the values of x and y in transaction *U* and a_{i0} , a_{j0} and a_{k0} , are the initial values of a_i, a_j and a_k

Interleaving 1

T	U
x:=read(j)	
	x:= read(k)
	write(i, 55)
y = read(i)	
	y = read(j)
	write(k, 66)
write(j, 44)	
write(i, 33)	

Interleaving 2

T	U
x:=read(j)	
y:=read (i)	
	x= read(k)
write(j, 44)	
write(i, 33)	
	write(i, 55)
	write(k, 66)

Interleaving 1 is equivalent to *U* before T. y_T gets the value of 55 written by *U* and at the end a_i =33; a_j =44; a_k = 66.
Interleaving 2 is equivalent to *T* before U. y_U gets the value of 44 written by *T* and at the end a_i =55; a_j =44; a_k = 66.

12.3 Give serially equivalent interleaving of *T* and *U* in Exercise 12.2 with the following properties: (i) that is strict; (ii) that is not strict but could not produce cascading aborts; (iii) that could produce cascading aborts.

12.3 Ans.

i) For strict executions, the *reads* and *writes* of a transaction are delayed until all transactions that have previously written the same objects are either committed or aborted. We therefore indicate the commit of the earlier transaction in our solution (a variation of B in the Answer to 12.6):

```
  T                      U
x:=read(j)
y:=read (i)
                   x:= read(k)

write(j, 44)
write(i, 33)
Commit
                    write(i, 55)
                    y:=read (j)
                    write(k, 66)
```

Note that U's *write(i, 55)* and *read(j)* are delayed until after T's commit, because *T write*s a_i and a_j.

ii) For serially equivalent executions that are not strict but cannot produce cascading aborts, there must be no dirty reads, which requires that the *reads* of a transaction are delayed until all transactions that have previously written the same objects are either committed or aborted (we can allow *writes* to overlap). Our answer is based on B in Exercise 12.2.

```
  T                      U
x:=read(j)
y:=read (i)
                    x:= read(k)

write(j, 44)
write(i, 33)
                    write(i, 55)

Commit
                    y:=read (j)
                    write(k, 66)
```

Note that U's *write(i, 55)* is allowed to overlap with T, whereas U's *read (j)* is delayed until after *T* commits.

iii)For serially equivalent executions that can produce cascading aborts, that is, dirty reads are allowed. Taking A from Exercise 12.2 and adding a *commit* immediately after the last operation of U, we get:

```
  T                      U
x:=read(j)
                    x:= read(k)
```

	write(i, 55)
y:=read (i)	
	y:=read (j)
	write(k, 66)
	commit
write(j, 44)	
write(i, 33)	

12.8 Explain why serial equivalence requires that once a transaction has released a lock on an object, it is not allowed to obtain any more locks. A server manages the objects a1, a2, ... *a*n. The server provides two operations for its clients:
read (i) returns the value of ai
write(i, Value) assigns *Value* to *ai*
The transactions *T* and *U* are defined as follows:
T: x= read (i); write(j, 44);
U: *write(i, 55);write(j, 66);*
Describe an interleaving of the transactions *T* and *U* in which locks are released early with the effect that the interleaving is not serially equivalent.

12.8 Ans.
Because the ordering of different pairs of conflicting operations of two transactions must be the same.
For an example where locks are released early:

T	T's locks	U	U's locks
	lock i		
x:= read (i);			
	unlock I		
			lock i
		write(i, 55);	
			lock j
		write(j, 66);	
		commit	unlock i, j

T conflicts with *U* in access to *a*i. Order of access is *T* then U.
T conflicts with *U* in access to *a*j. Order of access is *U* then T. These interleavings are not serially equivalent.

12.9 The transactions *T* and *U* at the server in Exercise 12.8 are defined as follows:
T: x= read (i); write(j, 44);
U: write(i, 55);write(j, 66);
Initial values of *ai* and *aj* are 10 and 20. Which of the following interleavings are serially equivalent and which could occur with two-phase locking?
SEE DIAGRAMS p. 512

12.9 Ans.
a) serially equivalent but not with two-phase locking.
b) serially equivalent and with two-phase locking.
c) serially equivalent and with two-phase locking.
d) serially equivalent but not with two-phase locking.

12.16 Consider optimistic concurrency control as applied to the transactions *T* and *U* defined in Exercise 12.9. Suppose that transactions *T* and *U* are active at the same time as one another. Describe the outcome in each of the following cases:

i) T's request to commit comes first and backward validation is used;
ii) U's request to commit comes first and backward validation is used;
iii) T's request to commit comes first and forward validation is used;
iv) U's request to commit comes first and forward validation is used.
In each case describe the sequence in which the operations of *T* and *U* are performed, remembering that writes are not carried out until after validation.

12.16 Ans.
i) T's *read(i)* is compared with *write*s of overlapping committed transactions: OK *(U* has not yet committed).*U* - no *read* operations: OK.
ii) *U* - no *read* operations: OK. T's *read(i)* is compared with *write*s of overlapping committed transactions *(*U's *write(i)):* FAILS.
iii)T's *write(j)* is compared with *reads* of overlapping active transactions *(*U): OK. U's *write(i)* is compared with *reads* of overlapping active transactions (none): OK *(T* is no longer active).
iv)U's *write(i)* is compared with *reads* of overlapping active transactions *(*T's *read(i*)): FAILS. T's *write(j)* is compared with *reads* of overlapping active transactions (none): OK.

12.17 Consider the following interleaving of transactions *T* and U:

T	U
openTransaction	*openTransaction*
y= read(k);	
	write(i, 55);
	write(j, 66);
	commit
x= read(i);	
write(j, 44);	

The outcome of optimistic concurrency control with backward validation is that *T* will be aborted because its read operation conflicts with U's write operation on *a*i, although the interleavings are serially equivalent. Suggest a modification to the algorithm that deals with such cases.

12.17 Ans.
Keep ordered read sets for active transactions. When a transaction commits after passing its validation, note the fact in the read sets of all active transactions. For example, when *U* commits, note the fact in T's read set.
Thus, T's read set = *{U* commit, i}
Then the new validate procedure becomes:

```
boolean valid = true
for (Ti = startTn + 1; Ti++; Ti <= finishTn) {
   let S = set of members of read set of Tj before commit Ti
   IF S intersects write set of T1
     THEN valid := false
}
```

12.18 Make a comparison of the sequences of operations of the transactions *T* and *U* of Exercise 12.8 that are possible under two-phase locking (Exercise 12.9) and under optimistic concurrency control (Exercise 12.16).
12.18 Ans.
The order of interleavings allowed with two-phase locking depends on the order in which *T* and *U* access *a*i. If *T* is first we get (b) and if *U* is first we get (c) in Exercise 12.9. The ordering of 12.9b for two-phase

locking is the same as 12.16 (i) optimistic concurrency control. The ordering of 12.9c for two-phase locking is the same as 12.16 (ii) optimistic concurrency control if we allow transaction *T* to restart after aborting. In this example, the sequences of operations are the same for both methods.

12.19 Consider the use of timestamp ordering with each of the example interleavings of transactions *T* and *U* in Exercise 12.9. Initial values of a_i and a_j are 10 and 20, respectively, and initial read and write timestamps are t_0. Assume each transaction opens and obtains a timestamp just before its first operation, for example, in (a) *T* and *U* get timestamps t_1 and t_2 respectively where $0 < t_1 < t_2$. Describe in order of increasing time the effects of each operation of *T* and U. For each operation, state the following:
i) whether the operation may proceed according to the write or read rule;
ii) timestamps assigned to transactions or objects;
iii) creation of tentative objects and their values.
What are the final values of the objects and their timestamps?

12.19 Ans.
a) Initially:
a_i: value = 10; write timestamp =max read timestamp = t_0
a_j: value = 20; write timestamp =max read timestamp = t_0
T: *x:= read (i*); *T* timestamp = t_1;
read rule: t_1> write timestamp on committed version (t_0) and Dselected is committed:
allows *read* x = 10; max read timestamp(a_i) = t_1. (see Figure 12.31a and read rule page 500)
U: *write(i, 55*); *U* timestamp = t_2
write rule: t_2 >= max read timestamp (t_1) and t_2> write timestamp on committed version (t_0):
allows *write* on tentative version a_i: value = 55; write timestamp =t. (See write rule page 499)
T: *write(j, 44*);
write rule: t_1 >= max read timestamp (t_0) and t_1 > write timestamp on committed version (t_0):
allows *write* on tentative version a_j: value = 44; write timestamp(a_j) =t_1
U: *write(j, 66*);
write rule: t_2 >= max read timestamp (t_0) and t_2 > write timestamp on committed version (t_0)
allows *write* on tentative version a_j: value = 66; write timestamp =t_2
T commits first:
a_j: committed version: value = 44; write timestamp =t_1; read timestamp = t_0

U commits:
a_i: committed version: value = 55; write timestamp =t_2; max read timestamp = t_1
a_j: committed version: value = 66; write timestamp =t_2; max read timestamp = t_0
b) Initially as (a);
T: *x:= read (i*); *T* timestamp = t_1;
read rule: t_1> write timestamp on committed version (t_0) and Dselected is committed:
allows *read*, x = 10; max read timestamp(a_i) = t_1
T: *write(j,44)*
write rule: t_1 >= max read timestamp (t_0) and t_1 > write timestamp on committed version (t_0):
allows *write* on tentative version a_j: value = 44; write timestamp =t_1
U: *write(i, 55*); *U* timestamp = t_2
write rule: t_2 >= max read timestamp (t_1) and t_2> write timestamp on committed version (t_0):
allows *write* on tentative version a_i: value = 55; write timestamp =t_2
U: *write(j, 66*);
write rule: t_2 >= max read timestamp (t_0) and t_2 > write timestamp on committed version (t_0):
allows *write* on tentative version a_j: value = 66; write timestamp =t_2
T commits first:
a_j: committed version: value = 44; write timestamp =t_1; max read timestamp = t_0
U commits:
a_i: committed version: value = 55; write timestamp =t_2; max read timestamp = t_1
a_j: committed version: value = 66; write timestamp =t_2; max read timestamp = t_0

c) Initially as (a);

U: *write(i, 55)*; *U* time stamp = *t*1;
write rule: *t1* >= max read timestamp *(*t0) and *t2* > write timestamp on committed version *(*t0):
allows *write* on tentative version *a*i: value = 55; write timestamp =*t1*
U: *write(j, 66)*;
write rule: *t2* >= max read timestamp *(*t0) and *t2* > write timestamp on committed version *(*t0):
allows *write* on tentative version *a*j: value = 66; write timestamp =*t1*
T: *x:= read (i)*; *T* time stamp = *t2*
read rule: *t*2> write timestamp on committed version *(*t0), write timestamp of Dselected = *t1* and Dselected is not committed: WAIT for *U* to commit or abort. (See Figure 12.31 c)
U commits:
*a*i: committed version: value = 55; write timestamp =*t*1; max read timestamp(*a*i) = t0
*a*j: committed version: value = 66; write timestamp =*t*1; max read timestamp(*a*j) = t0
T continues by reading version made by *U* x = 55; write timestamp(*a*i) =*t*1; read timestamp(*a*i) = *t2t2*
T: *write(j,44)*
write rule: *t2* >= max read timestamp *(*t0) and *t2* > write timestamp on committed version *(t*1):
allows *write* on tentative version *a*j: value = 44; write timestamp =*t2*
T commits:
*a*i: committed version: value = 55; write timestamp =*t*1; max read timestamp = *t2*
*a*j: committed version: value =44; write timestamp =*t*2; max read timestamp = t0

d) Initially as (a);
U: *write(i, 55)*; *U* time stamp = *t*1;
write rule: *t1* >= max read timestamp *(*t0) and *t*2> write timestamp on committed version *(*t0)
allows *write* on tentative version *a*i: value = 55; write timestamp(*a*i) =*t1*
T: *x:= read (i)*; *T* time stamp = *t2*
read rule: *t*2> write timestamp on committed version *(*t0), write timestamp of Dselected = t1 and Dselected is not committed: Wait for *U* to commit or abort. (See Figure 12.31 c)
U: *write(j, 66)*;
write rule: *t2* >= max read timestamp *(*t0) and *t2* > write timestamp on committed version(t0)
allows *write* on tentative version *a*j: value = 66; write timestamp =*t1*
U commits:
*a*i: committed version: value = 55; write timestamp =*t*1; max read timestamp = t0
*a*j: committed version: value = 66; write timestamp =*t*1; max read timestamp = t0
T continues by reading version made by U, x = 55; max read timestamp(*a*i) = *t2*
T: *write(j,44)*
write rule: *t2* >= max read timestamp *(*t0) and *t2* > write timestamp on committed version *(t*1)
allows *write* on tentative version *a*j: value = 44; write timestamp=*t2*
T commits:
*a*i: committed version: value = 55; write timestamp =*t*1; max read timestamp = *t2*
*a*j: committed version: value = 44; write timestamp =*t*2; max read timestamp = t0

12.21 Repeat Exercise 12.20 using multiversion timestamp ordering.
12.21 Ans.
The main difference for multiversion timestamp ordering is that *read* operations can use old committed versions of objects instead of aborting when they are too late (see Page 502). The read rule is:
let Dselected be the version of D with the maximum write timestamp <= Tj

IF Dselected is committed THEN
    perform *read* operation on the version Dselected
ELSE *Wait* until the transaction that made version Dselected commits or aborts

write operations cannot be too late, but writes are checked against potentially conflicting *read* operations. The write rule is taken from page 402. Recall that each data item has a history of committed versions.

For the ordering on the left of Exercise 12.20, we show that the outcome is the same. Timestamps are T: *t1* and U:*t*2. Initial state as in Exercise 12.9 a. Call these committed versions Vt0.
U: *write(i, 5*5);
write rule: read timestamp of DmaxEarlier t0 <=*t2*
allows *write* on tentative version *a*i: value = 55; write timestamp =*t2*
U: *write(j, 66*);
write rule: read timestamp of DmaxEarlier t0 <= *t2*
allows *write* on tentative version *a*j: value = 66; write timestamp =*t2*
T: *x:= read (i*);
Select version with write timestamp t0 *read* x = 10; its read timestamp becomes *t1*
T: *write(j,44)*
write rule: read timestamp of DmaxEarlier *t1* <= *t1*
allows *write* on tentative version *a*j: value = 44; write timestamp =*t1*
U commits:
*a*i: committed version Vt2: value = 55; write timestamp =*t*2;
*a*j: committed version Vt2: value = 66 and write timestamp =*t2*
T commits:
*a*j: committed version Vt1: value = 44; write timestamp =*t1*
For the ordering on the right of Exercise 12.20, we show that the outcome is different in that *T* does not abort.
It proceeds in the same way as the first until *U* has performed both its *write* operations and commits. At this stage we have:

*a*i: committed version Vt2: value = 55; write timestamp =*t2*
*a*j: committed version Vt2: value = 66; write timestamp =*t2*
T: *x:= read (i*);
The *read* selects the committed version Vt0 and gets x=10 and the read timestamp = *t1*
T: *write(j,44)*
write rule: read timestamp of DmaxEarlier *t1* <= t1
allows *write* on tentative version. *a*j: value = 44; write timestamp =*t2*
T commit:
*a*j: committed version Vt1: value = 44; write timestamp =t1

12.23 What are the advantages and drawbacks of multiversion timestamp ordering in comparison with ordinary timestamp ordering?

12.23 Ans.
The algorithm allows more concurrency than single version timestamp ordering but incurs additional storage costs.
Advantages:
The presence of multiple committed versions allows late *read* operations to succeed.
write operations are allowed to proceed immediately unless they will invalidate earlier reads (a *write* by a transaction with timestamp *Ti* is rejected if a transaction with timestamp *Tj* has read a data item with write timestamp *Tk* and *Tk* < *Ti* < *T*j).

Drawbacks:
The algorithm requires storage for multiple versions of each committed objects and for information about the read and write timestamps of each version to be used in carrying out the read and write rules. In the case that a version is deleted, *read* operations will have to be rejected and transactions aborted.
Exercise 13.15 shows that the algorithm can provide yet more concurrency, at the risk of cascading aborts, by allowing *read* operations to proceed immediately. In this case, to ensure recoverability, requests to commit must be delayed until any the completion (commitment or abortion) of any transaction whose tentative objects have been observed.