This page is intentionally left blank.

Chapter 13

Integrity of Databases

Integrity is praised, and starves.

Juvenal
Satires, vol. 1 (A.D. 110)

Throughout the previous chapters we have implied that a well-structured database system, using careful data entry and update procedures, and operating reliably and free from unauthorized access, will be a secure depository for the data. There is, however, one area where failures can be generated while everything else seems to be proceeding satisfactorily. This problem area exists because of interference among the multiple transactions which are active within one computer system. The interference can be related to competition as well as to cooperation.

Integrity of a database means absence of inconsistent data. While an update transaction is in progress, the consistency of a database can be temporarily disturbed. One data element will reflect the update, while another element has not yet been changed. A low level of redundancy reduces potential integrity violations, since one fact will appear only rarely in more than one place. We have designed our databases with that objective, and yet find that they contain redundancies.

An error in a single transaction can cause a failure if a database update is not carried to completion, but the techniques presented in Secs. 11-3 and 11-4 deal with this problem successfully. However, even error-free update transactions can cause problems when they access shared data in a time period where there is other activity. Accessors sharing the data can pick up temporary inconsistencies caused by a concurrent update and spread the inconsistency through the database or to the output sent to the users.

Redundancy In order to achieve rapid access we create structures which are redundant. We also maintain connections between relations which employ redundancy. Less obvious are redundancies due to derived data, and due to externally imposed integrity constraints.

An index is a redundant access structure. The department name appears in each employee record and in one record of the department relation. The departmental salary expense is equal to the sum of all its employees' salaries. The total number of employees on the payroll should not change when an employee transfers from one department to another.

Many of the issues of integrity maintenance are also of concern to developers of operating-system services. We will in fact assume the availability of basic operating–system facilities and restrict our discussions to the interactions caused by database usage. An operating system will concentrate on controlling the competition for system resources due to interfering processes.

Competition Performance degradation due to competing multiple processes was seen in Sec. 5-4, and in Sec. 6-3 multiple processes were spawned by a single computation so that devices could be activated independently. Computations within a system may remain conceptually independent, and yet affect each other.

For databases we will deal again within the framework of *transactions*, as defined in Secs. 1-7-5 and 11-3-1. Although transactions may spawn multiple processes, we have to consider only cases of interference among processes which are initiated independently or, equivalently, by different transactions. We recall that a transaction is a computation that is initiated by a user, allowed to complete as rapidly as the available hardware permits, and then terminated.

The primary task of a transaction-oriented operating system is to keep one transaction as much as possible unaffected by the presence of other transactions. This independence is not completely realizable. There will be competition for the use of processors (especially if there is only one), for the use of disks and the channels to access them, and for the use of devices for input and output. Older devices for input and output (tape drives, readers, printers) are especially awkward to share since manual intervention is required to select the tapes to be mounted and to keep the paper going in and out in the proper order.

In a database operation there are additional resources to be shared. All users will want to access the schema, many may select a particular file, or index, or hierarchical level, and several may wish to access the same data item. Data are perfectly sharable, until one or more transactions want to make changes. Then some users may be denied access to the data to avoid the dissemination of inconsistent information; Figs. 13-1 and 13-2 will illustrate the problem. Now competition for **data** resources ensues.

Cooperation When a transaction has spawned multiple processes, their activities have to be coordinated (see Fig. 6-9). These processes may consist of multiple sections and spawn new processes themselves. In Example 8-8 database procedures were initiated to create `ACTUAL` results in storage, after an update or insertion of new source data in a file. Often transactions initiated by distinct users may have to be similarly coordinated.

An example where explicit cooperation is required can be found in the generation of salary checks by a payroll department. The transaction should be executed only after all `hours_worked` and salary raises have been entered for the period.

Locking, as described in Sec. 13-1 below, provides the means to keep transactions from interfering with each other. Locking is presented using examples of the various cases to be considered and the mechanisms which can deal with them. All the mechanisms have their own liabilities and must be considered with care. An especially serious problem due to locking is the possibility of mutual lock-out or deadlock, the topic of Sec. 13-2. We review data integrity in Sec. 13-3.

13-1 LOCKING

The basic problem of competition for apparently sharable resources can be illustrated by the interaction of the two transactions illustrated in Fig. 13-1. When more than one transaction obtains a copy of some data element for updating, the final database value for that element will be the result of the last transaction to write the updated value. The result of the other transaction has been overwritten. The database does not reflect the intended result, here `230 Widgets`.

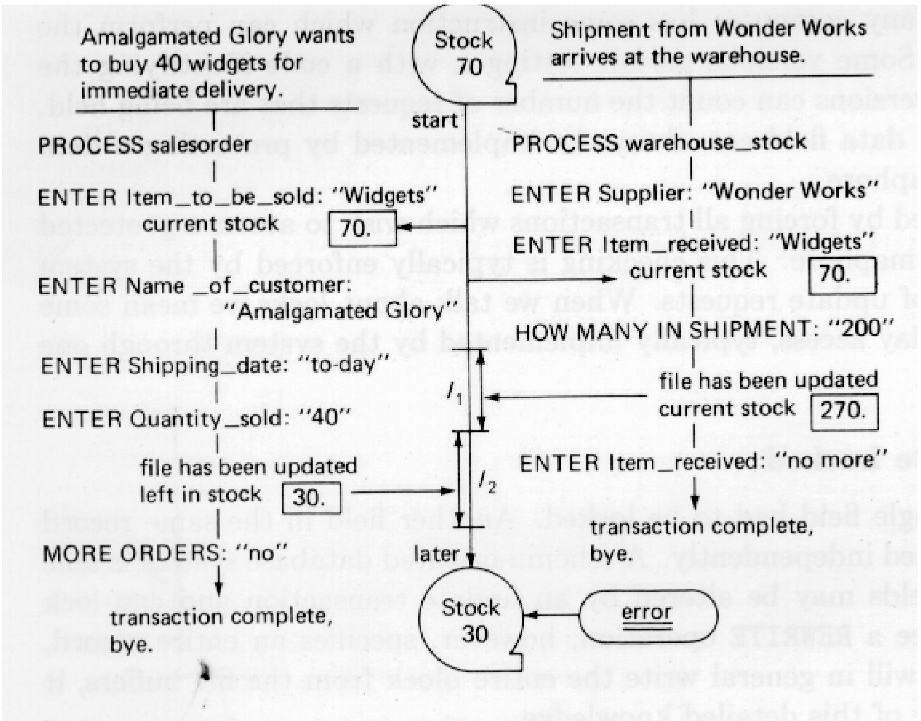
Amalgamated Glory wants to buy 40 widgets for immediate delivery.
Stock 70
Shipment from Wonder Works arrives at the warehouse.
start
PROCESS salesorder
PROCESS warehouse_stock
ENTER Item_to_be_sold: "Widgets"
current stock 70.
ENTER Supplier: "Wonder Works"
ENTER Item_received: "Widgets"
current stock 70.
ENTER Name_of_customer: "Amalgamated Glory"
HOW MANY IN SHIPMENT: "200"
ENTER Shipping_date: "to-day"
I_1
file has been updated
current stock 270.
ENTER Quantity_sold: "40"
ENTER Item_received: "no more"
file has been updated
left in stock 30.
I_2
transaction complete, bye.
MORE ORDERS: "no"
later
transaction complete, bye.
Stock 30
error

Figure 13-1 Update interference.

The integrity of results must also be protected. Figure 13-2 shows a case where a read transaction, executed while an update is in progress, produces an erroneous result. We also use Fig. 13-2 to illustrate other aspects of locking.

The solution to integrity problems is the provision of a *locking* mechanism which assures mutual exclusion of interfering transactions. The transaction which is the first to claim the nonsharable object becomes temporarily the owner of the object. Other transactions which desire access to the object are blocked until the owner releases the claim to the object.

13-1-1 A Lock Mechanism

The claim to an object is made by setting a *semaphore*. A simple binary semaphore is a protected variable `x`, which indicates if the associated resource is `free` or `occupied`. A semaphore can be set to indicate `occupied` only if it indicated before that the resource was `free`. The transaction which sets the semaphore becomes its owner, and only it is permitted to reset the semaphore to `free`.

In order to implement a semaphore which can be tested and set in this manner we need a primitive, indivisible computer operation. A typical instruction is shown in Example 13-1.

Example 13-1 Semaphore Machine Instruction

Label	Operation	Address	Comment
`alpha`	`TEST_AND_SET`	`x`	*If* `x=0` *set* `x:=1` *and skip to* `alpha+2`;
	`JUMP`	`to_wait`	*otherwise continue at* `alpha+1`
	`OPEN`	`resource_x`	*Go on and grab the resource*

Waiting may require calling the system for later scheduling of this transaction; scheduling and queuing was described in Sec. 6-3.

We assume that any computer has some instruction which can perform the semaphore function. Some versions permit setting `x` with a code identifying the current owner; other versions can count the number of requests that are being held. An identifier or count data field can always be implemented by protecting a data field itself with a semaphore.

Locking is achieved by forcing all transactions which wish to access a protected object to check the semaphore. This checking is typically enforced by the system during the execution of update requests. When we talk about *locks* we mean some means to limit and delay access, typically implemented by the system through one or several semaphores.

13-1-2 Objects to Be Locked

In Fig. 13-1 only a single field had to be locked. Another field in the same record could have been modified independently. A schema-oriented database system is able to know what data fields may be altered by an update transaction and can lock only those fields. Since a `REWRITE` operation, however, specifies an entire record, and since file systems will in general write the entire block from the file buffers, it is difficult to make use of this detailed knowledge.

Granularity The choice of size of objects to be locked is referred to as the *granularity* of locking. The smallest unit which can be identified for locking through a schema is a field or a segment as defined in Sec. 8-3-1; for file access it is a record, and the smallest unit available for buffer management is typically a block. Many operating systems provide locks only for entire files.

Having small granules will considerably enhance system performance since the probability of interference will be less. For read operations we can manage small granules; for update we may be limited to blocks. Small granules means having more locks, so that they are more difficult to manage. The management of locks is presented in Sec. 13-1-7. All granules have to be positively identified so that, if they are locked, they will be associated with only one lock.

Identification of Objects The identification of the granules, for instance, records, is a function of some supervisory system. Each granule most be positively identified. The users may not be able to recognize synonymous record names and will be even less aware about any shared blocks.

Example 13-2 Identifying an Object

In a complex file structure, the same record may be accessed via multiple paths, and hence may seem to have multiple names. Three alternate path-names for the data element which contains the inventory count of `Widgets`, stored in a warehouse for a customer may be:

```
supplier = "Wonder Works",  part = "Widget";
warehouse = "Centreville",  shelfno = "321005";
customer = "Grandiose Co",  account_item = 14.
```

One candidate for a unique identification is the TID, presented in Sec. 4-2-3. If the lock applies to blocks, the relative block number provides the required identification. To identify fields a catenation `block_number.record_number.field_number` may be used. The record number in a block is found through the marking scheme shown in Fig. 2-11 and the field number may be based on the schema entry. When a record spans blocks, multiple blocks have to be managed as one object.

Lock Interval Locking also has a time dimension. A protective approach is to lock each object read or needed for later update as early as possible. When the transaction is completed all locks will be freed.

The lock interval can be shortened by releasing locks from an object as soon as the update is performed. This means that each object to be read has to be locked if an update is contemplated. Locking during the `READ_TO_UPDATE` – `REWRITE` interval is typical when using the IMS system; the three `GET` operations seen in Sec. 9-6-2 have the alternate forms shown in Table 13-1.

Table 13-1 IMS Locking Commands

The operations
```
    Get_and_Hold_Unique  segment_sequence
    Get_and_Hold_Next    segment_sequence
    Get_and_Hold_Next_within_Parent  segment_sequence
```
will lock the `segment_sequence` *until a writing, that is a*
    `REPLace` *or* `DeLETe`
operation on the `segment_sequence` *is executed.*

A transaction may of course need to update several objects. Holding several records, updating and releasing them when done violates the rules defined for protection when using the two-phase transaction concept, introduced in Sec. 11-4. No records should be written before the commit point. When the commit message is given to the transaction manager and accepted all writing takes place.

This means that the transaction locks data resources up to the commit point. Resources held by one user are not available to others. To get a high level of usage from a computer system it is desirable to exclude other users for as short a time as possible.

In the example of Fig. 13-1 the resource will have to be held for at least several seconds, since the READ and REWRITE operations are separated by an intermediate terminal activity. The time that other users are excluded can be reduced if we do not lock until ready to update the file. But the number of Widgets can be different now. Since within the transaction processing interval (I_1 or I_2 in Fig. 13-1), the Widgets in stock may have been sold by another transaction, a new READ_TO_UPDATE with a lock is executed before the REWRITEs. The value obtained by the second READ is checked again and, if found different and perhaps inadequate, the customer has to be informed and the transaction aborted. Data which dont contribute to the result do not have to be reread. The customer may have asked about other items, or there may been a check about previous Widget orders.

Deferring locking Extending the two-phase transaction concept to locking we can avoid actually locking resources until the transaction is completely defined. Not locking means there is some chance of another transaction having modified the data so that all data which contribute to the result have to be reread. When the transaction is ready to be committed all data are reread with locks. The time that other users are excluded is reduced, but at a cost: each record has to be read twice. If the result of the new read is not identical the transaction has to *release* the commit and redo its computation from the beginning. This mode of operation also has implications in avoiding deadlock, as will be discussed in Sec. 13-2-3.

Locking the devices to reduce the lock interval The cost of an extra READ_TO_UPDATE just prior to a REWRITE is small since little seek interference is expected. Even if only one READ is performed the lock interval may be much smaller if no other transactions access the devices being used and seeks are avoided. The assumption that seek interference is negligible has been made throughout the derivation of performance formulas (Eqs. 2-29 to 2-31). A transaction may be able to impose such a condition via a lock.

The possibility of a seek between READ_TO_UPDATE and the REWRITE can be avoided by letting the lock, which is set at the time of the READ_TO_UPDATE operation not only exclude other accesses to the object, but also exclude all other accesses to the entire device. In programming terminology the sequence of instructions between READ_TO_UPDATE and the completion of the REWRITE is made into a *critical section*.

13-1-3 Locking to Protect Transaction Results

We considered up to this point locks to protect the database. A temporary inconsistency introduced by an updating transaction can also create erroneous answers in transactions which report answers to the users. The example of Fig. 13-2 gives a scenario leading to a wrong result. This case appears not to be as serious, since the database will not be left in an erroneous state.

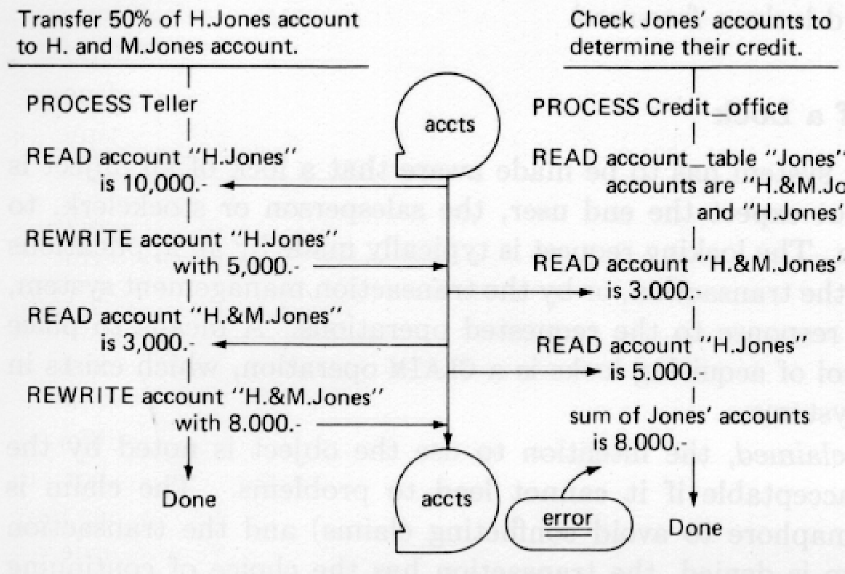
Transfer 50% of H.Jones account to H. and M.Jones account.
PROCESS Teller
READ account "H.Jones" is 10,000.-
REWRITE account "H.Jones" with 5,000.-
READ account "H.&M.Jones" is 3,000.-
REWRITE account "H.&M.Jones" with 8.000.-
Done
accts
accts
Check Jones' accounts to determine their credit.
PROCESS Credit_office
READ account_table "Jones" accounts are "H.&M.Jones" and "H.Jones"
READ account "H.&M.Jones" is 3.000.-
READ account "H.Jones" is 5.000.-
sum of Jones' accounts is 8.000.-
error
Done

Figure 13-2 Read interference.

The Jones's can probably set the record straight later if they get a lower than expected credit rating. If they are familiar with computing, they may realize that the poor rating was due to confusion created by other requests they made to their bank.

However, a system which produces erroneous results with any frequency will soon lose the confidence of the users. Locking is hence also appropriate in this case. Since no REWRITE will be given an explicit UNLOCK may be needed to release a lock once it is set. The update transaction should already be locked to prevent update interference. Read-only transactions which need to be correct, i.e., have to be able to stand up to an *audit*, should issue locks against simultaneous updates.

We note that erroneous results are only created due to interfering updates which modify multiple related data elements. In this example the lock interval extends over several update operations, but no intervening terminal operations occur in the interval.

Audit-reads and Free-reads While correct results from read-only queries are always preferable, we can envisage queries which do not require lockout of concurrent updates. A READ NOLOCK statement is available in PL/1 for that purpose. Queries which only fetch a single data value do not require protection by locks, although the single answer may be slightly out of date. If up-to-dateness is a problem, a second data element which contains the update time has to be fetched, transforming the single-element query to a multiple-element query. Now a decision to lock or not to lock has to be made.

Inquiries leading to a summarization of many values may also avoid the expense of locking and disabling concurrent updates and be allowed to proceed freely, and not test or set locks. If many values are aggregated in the query result, the effect of errors due to interference should be negligible. The use of locks to ensure audit-proof results to perform tabulations which require many accesses to a file can make shared database usage impossible. Audit-reads can be always be performed in systems which use versioning (Sec. 11-3-3) since the data elements will not be shared between reading and writing. We will call a query which needs to lock an *audit-read* and a query which can disregard locks a *free-read*.

13-1-4 Specification of a Lock

The database supporting system has to be made aware that a lock of an object is requested. One would not expect the end user, the salesperson or stockclerk, to request the locking action. The locking request is typically made by an applications programmer who defines the transaction, or by the transaction management system, or by the file system in response to the requested operations. A means to place the programmer in control of acquiring locks is a `CLAIM` operation, which exists in various forms on many systems.

When an object is *claimed*, the intention to use the object is noted by the system. The claim is acceptable if it cannot lead to problems. The claim is noted (using again a semaphore to avoid conflicting claims) and the transaction can proceed. If the claim is denied, the transaction has the choice of continuing with another task, maybe to return later, or to be placed into a waiting queue. Section 6-3 presented some examples of such queues.

When claimed objects are accessed then locks are set. There may be delays, but eventually a transaction waiting in the queue for an object is granted access to the claimed object and can proceed. The transaction sets a lock to gain exclusive access to the object, which will be held until the transaction executes a `RELEASE` command. Claims held by a transaction are also released when the transaction completes or aborts.

Transactions which terminate without releasing their locks are considered to be defective. If the system has the capability to undo the effects of defective transactions and to restore the database (Sec. 11-4), such a restoration is recommended. At a minimum the system administrator has to be informed when transactions halt without releasing all of their locks on resources. If locks are not released a part of the system becomes unavailable.

Table 13-1 showed some instructions which specify locks for segments of a hierarchical database file. When using operating systems other than transaction systems the control language may give a choice if a specific file can be shared among programs or cannot be shared. To prevent sharing a lock is associated with an entire file. The system will disallow any shared use of the file once it has been opened until it is again closed. Since `OPEN` and `CLOSE` operations tend to be costly, the lock interval tends to be long. The operating system, rather than the program controls any locking. For instance, the PL/1 statement

```
    OPEN FILE stock UPDATE
```

does not provide integrity protection. It permits `READ, REWRITE` sequences to be interrupted, and since execution of a `REWRITE` is not required no cue other than a `CLOSE` operation is available to release the object.

If sharing is permitted by the operating system, the programmed transactions or a transaction manager assumes some responsibility. The file, when it is initially defined or opened, is characterized to accept certain operations. The statement

```
    OPEN FILE(stock) EXCLUSIVE UPDATE
```

of PL/1 will cause any `READ` statement to set a lock on the record being read. Only one lock can exist per file. That lock will remain until either a `REWRITE`, a `DELETE`, or an explicit `UNLOCK` statement for that record is executed.

In systems using virtual memory support for files, some automatic protection exists for pages being updated. Either a single page will be shared among multiple processes, or the fact that a copy has been modified will force subsequent accessors to fetch the modified page. Problems can still occur if pages are modified, as was the case in Fig. 13-1, from data which are the result of calculations in the private region of a user. Since the paging system protects the access to the files, only the pages in memory need to be protected. If users share single copies of storage pages in memory, interference protection becomes simple and rapid. The transactions can set locks on the pages in memory, do the reread, check for interference, and, if the read values are still the same, do rewrite operations and release the locks. The same technique works if a file buffer managing program keeps only a single copy of each block in memory.

Regions to Be Locked Example 13-1 illustrated an interference problems due to concurrent updating of a single atomic object. While this problem certainly requires resolution, other interference problems may require locking of collections of objects, or *regions*.

A simple example was shown in Fig. 13-2, where data obtained by a read transaction are inconsistent because of a sequence of updates required to carry out a transfer of funds. In order to guarantee the correctness of other transactions a region comprised of two accounts (`H.Jones, H.&M.Jones`) has to be locked by the update transaction.

Regions have to be locked to avoid interference both from read and from update transactions. Figure 13-2 illustrated read interfence. A similar `Credit_office` transaction which computes and writes the credit rating of the `Jones`' would place an error into the database.

If very many or very large objects are included in the region to be locked for a transaction, the probability of other transactions being affected increases rapidly. In Sec. 13-1-5 we will discuss the extent of interference and in Sec. 13-1-6 the effect on performance. Section 13-1-7 will consider the management of locks.

The technique of holding objects until the update statement is executed fails when regions have to be considered. Regions either have to be defined by the programmer of the transaction or all resources accessed have to be held until the transaction is completed.

13-1-5 Interactions among Locks

We showed that it is necessary to provide lockouts among update transactions and between read-only queries which need results which can stand up to an audit and update transactions. Read-only queries do not interfere with each other. The use of lockouts makes a part of the database unavailable.

We also distinguished in Sec. 13-1-3 queries which needs to lock (audit-reads) and queries which can disregard locks (free-reads). The interaction rules are summarized in Table 13-2.

Blocking is achieved by having a locking semaphore set on the object of the operations. When the semaphore is freed the transaction scheduler can restart transactions which are queued because of that semaphore. The variable `flag` is a second semaphore to avoid *hibernation* of update transactions. This issue is discussed in Sec. 13-2-1.

Table 13-2 Data Access Interference

	Operation in Progress		
Request	`Update`	`Audit-reads`	`Free-reads`
`Update`	Block	Block and set `flag` on	Pass
`Audit-read`	Block	Pass unless `flag` on	Pass
`Free-read`	Pass	Pass	Pass

Serializability We have shown examples of interference among transactions. No interference would occur if each transaction would wait until its predecessor is finished. Such a schedule is called *serial*. The system performance, however, would be greatly impaired, since no computational or device overlap could occur. We wish to permit any overlap of transactions which will still give the correct answers and leave the database in some correct state. This will be true if the execution order of the primitive operations of the set of overlapping transaction gives a result which is equivalent to some *serial schedule* of the original transactions. The property of overlapping transactions having a corresponding serial transaction schedule is called *serializability*.

If we are conservative we assume that any transactions which share data are always sensitive to their excecution sequence, as defined in Sec. 11-3-4. Then we can consider only four cases of operations from two transactions on a shared data item.

Table 13-3 Cases for Serializability

Transactions A and B may each read or write the same data item `X`:

case / transaction	`rr`	`rw`	`wr`	`ww`
A:	`read X`	`read X`	`write X`	`write X`
B:	`read X`	`write X`	`read X`	`write X`
rearrangable	yes	`no`	`no`	yes

Serializability will be maintained if the operation sequence of any `rw` and `wr` cases is not interchanged. The `rr` case is not sensitive to order. In the `ww` case the value created by an overlapping transaction A is overwritten by transaction B and can be ignored if B has arrived. This case will occur rarely, since it actually indicates a poor transaction design. Transactions which independently write the same data element, without reading it first, to a file will produce results dependent on their relative time of execution. In practice some synchronization is required. There will be read operations to data or to some shared semaphore to control execution sequences.

Interference of Access Path Objects Access to intermediate objects often precedes access to goal data. Depending on the structure of the files in the database objects in the access path have to be locked as well. Typical access path objects are indexes, rings, or ancestors in a hierarchy. The access path can also involve intermediate data items, for instance, the `Department` record to find the `Employee` who is a `"Manager"`.

In the locking of access paths a hierarchical structure is prevalent.

A transaction has to assign a task to an employee with suitable skills. While the task record is being analyzed any employee record is a candidate for further locking. When the required skills have been determined only the records of employees with those skills have to be held. When the appropriate employee is found only that one employee record has to be locked for an update.

If an update requires the change of one or more indexes, subsidiary index levels and records have to be locked out from concurrent updates. Audit-reads and free-reads can be permitted to make progress only if indexes and pointers to goal data remain valid. For audit-reads the goal object still requires locking.

Update transactions in a hierarchical structure require a lockout of all descendants. Audit-reads are also affected because goal data are qualified by the data values of their ancestors. Unless rings are updated very carefully, it may be wise to also lock out free-reads. When a tree is being updated, the possibility of rebalancing can make all or a large portion of the tree (a broom of the tree) inaccessible. If rebalancing is infrequent per update, it is advantageous to treat the rebalancing algorithm as a separate process. This avoids excessively large claims associated with update transactions which may cause rebalancing.

Table 13-4 summarizes the additional access constraints. It is important to note that the regions to which access is blocked are much larger than the goal objects considered in Table 13-2.

Table 13-4 Access Path Interference

	Operation in Progress		
Request	`Update`	`Audit-reads`	`Free-reads`
`Update`	Block	Pass if safe	Pass if safe
`Audit-read`	Pass if safe	Pass	Pass
`Free-read`	Pass if safe	Pass	Pass

The qualifier, *if safe*, has to be based on a thorough understanding of the file update transactions and of the sequence in which operations are scheduled. Section 13-1-6 illustrates some cases. Buffer queuing algorithms which reorder processing sequences to achieve optimal device utilization can also affect the safety of interacting transactions.

Explicit Claiming of Regions If the system does not support two-phase protocols, or if the transactions are so extensive that it is not acceptable to lock the required resources for the length of the transaction, explicit claiming becomes necessary. It can also be difficult to deal automatically with a wide variety of resources: records, blocks, rings, pointer-lists, files, devices, etc.

We use again the operating system concept of a *critical section* (Sec. 1-2-2). If all the objects comprising the region to be accessed are known at the beginning of the critical section, a multiple object claim can be presented to the system. A claim for multiple objects itself has to be processed as a single critical section by the supporting operating system to avoid claim conflicts from other transactions. The operating system can again use a semaphore to defer other requests.

The allocation of the object is not necessarily carried out when the object is claimed. Many objects are claimed because they may be of interest, but will not be actually needed. When a claim is granted, the system only promises to make the object available when needed. This allows a transaction which desires access to the claimed object, but which will not otherwise interfere with another transaction, to proceed. Deferral of allocation to the point of actual use can significantly decrease congestion in a system which requires regions to be claimed at one single time.

Incremental Claims It is not always possible, at the initiation of the transaction, to specify all objects which may be needed eventually. Intermediate analysis of data can define additional objects which have to participate in the claim. Some incremental claims may be avoided by initially claiming large regions, at the cost of making much of the database inaccessible. A capability to claim objects incrementally can hence be necessary in order to provide flexibility in transaction processing or in order to achieve adequate system performance.

When claims are submitted incrementally and a later claim is denied, the objects claimed previously by the transaction are tied up in an unproductive manner. Incremental claims also require extensive lock management.

13-1-6 Performance Issues of Locking

The use of locks has a great impact on database performance. Locks set by one transaction make the database unavailable to many other transactions. We already defined, in Sec. 13-1-3, the concept of a `Free-read` in order to create a class of uninvolved transactions.

The degree to which performance is affected depends on the size of the object or region being locked and the length of the locking interval. In Sec. 13-1-2 we considered reduction of the lock interval in a transaction by rereading the input data for a transaction, with a lock, just prior to the rewrite for the update operation. The cost incurred here is a duplicate read operation. A quantitative evaluation has to estimate the delays due to locking, using estimates of load, transaction lengths, and granularity versus the increase of read operations.

Deferred File Update The use of locks during daily processing can be avoided if file modifications, made due to updates, are not immediately reflected in the files. When updating is deferred the blocks or records generated by update transactions are identified relative to the main database, but held on an update-log file. The update-log file is merged into the database during periods of no or little simultaneous activity, typically at night. Implicit in this approach is a disregard of issues of serializability: read operations do not reflect recent writes.

Deferred update can be necessary in systems where locks are restricted to large objects (files) while update involves small objects (records or fields).

Reducing the Object Size There may also be opportunities to reorganize the granularity of a transaction to reduce the locking requirements. The redundancy existing in files permits breaking up of some complex operations, and this reduces extensive locking requirements.

‡‡‡‡‡ Transactions which access many records of a file should be analyzed for granularity. In most systems the cost of claiming and releasing all file records individually is high. peformance can b improved by locking of an entire file, but now sharing the file is made impossible; locking blocks may be optimal.

The critical section required to protect `Jones` in Fig. 13-2 extends from the first `READ` to the last `REWRITE` of the transaction, and the region to be locked involves two objects. In complex queries the region which has to be locked can involve many objects:

```
DEDUCT taxes FROM Employees.pay AND PUT SUM INTO tax_account.
```

Analysis shows that only one `Employee` record at a time and the one ledger record, namely `tax_account`, has to be locked.

In order to reduce the size of the large locked object, the `Employee` file, locks are set and released within the loop. Example 13-3 indicates the points with †. The number of `Employee` records specified by the `DO`-loop should be remain fixed. Setting a read-lock on `no_of_employees` will prevent insertion and deletion of entire `Employee` records, but other values in the file remain fully accessible during the `Withhold_tax` transaction.

Example 13-3 A Transaction with Locking Alternatives

```
 Withhold_tax: TRANSACTION;
   tax_account = 0;
     DO FOR no_of_employees;
†        net_salary = gross_salary - tax;
         tax_account = tax_account + tax;        †
     END;
```

We note that the need to lock is again related to the existence of redundant derived information. The transitive functional dependency between an `employee`, `tax`, `gross_salary`, and `net_salary` is not obvious enough to be captured by a normalized view model. ‡‡‡‡‡

Reliability and Performance of Simultaneous Transactions Interference in access paths can cause transaction programs to fail. Moving an object can cause a transaction trying to locate the object via pointer to pick up invalid data or, worse, garbage when further pointers are expected. These problems can be minimized or entirely avoided by careful design of the procedures which manipulate the paths.

When access methods use indexes, obsoleted pointers can be kept safe by not reusing the space that they point to, although the space is freed when records are deleted or updated. To avoid fetching bad data a *tombstone* is placed in the old spot. The space becomes safely reusable when a reorganization can assure that all pointers to this spot are gone. In Sec. 3-4-1 we considered how locking for update protection can be minimized in B-trees.

In ring structures (Sec. 3-6-1) the problems of losing a path through the database due to a simultaneous update become even more pronounced. Figure 13-3 sketches a problem which can occur with any form of linked records. If it is possible for one user to read the file while another user, or a system task, updates the chain, care has to be taken that the reader of the chain sequence is not thrown off the track, as indicated by the stars in the diagram. The safe procedure shown on the right of Fig. 13-3 unfortunately tends to use more buffers and more access time to execute, since READ and REWRITE operations for one block are not paired. If the entire ring is locked the left choice does become safe, the performance of the specific transaction increases, but other, simultaneous transactions may be held up.

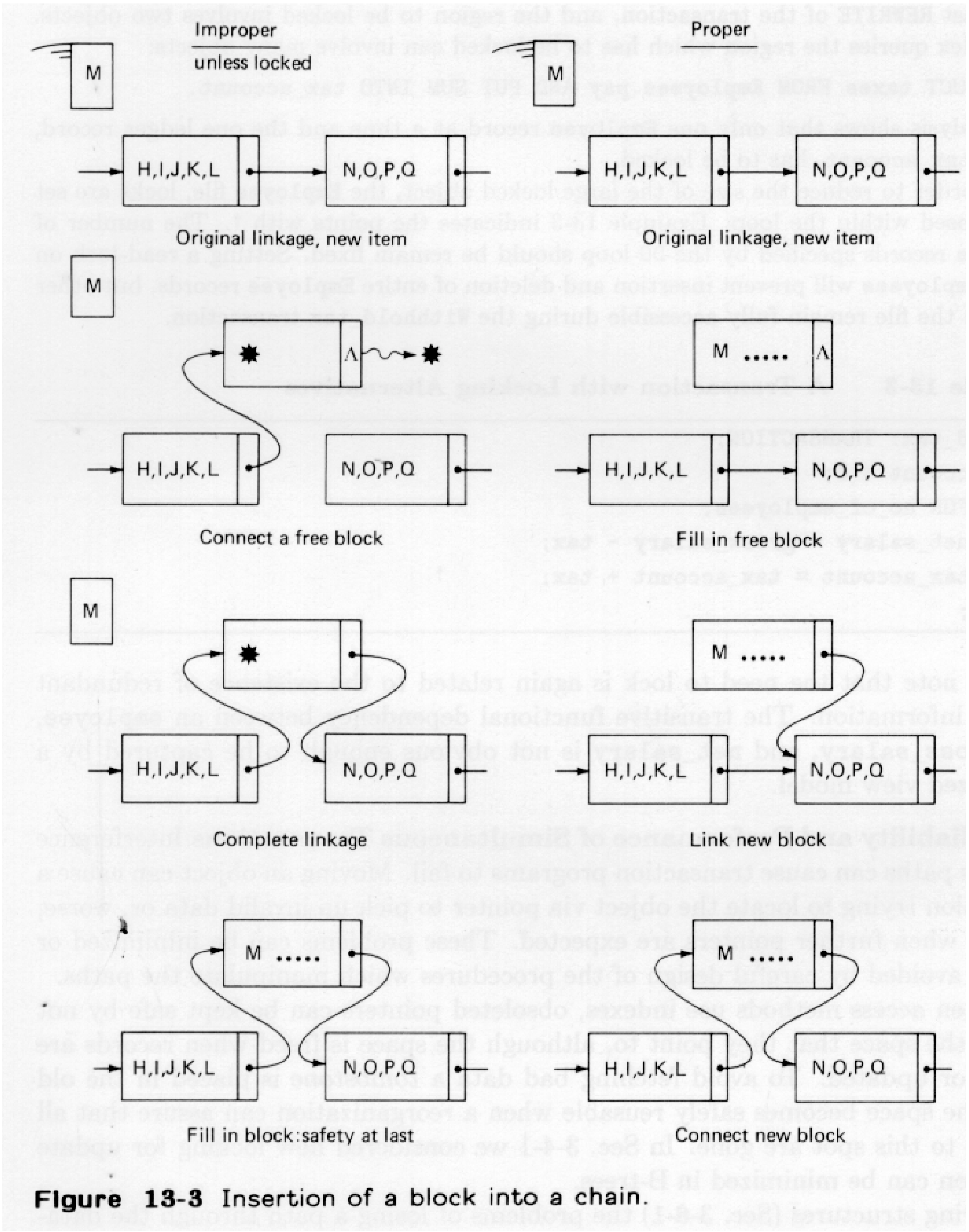

Figure 13-3 Insertion of a block into a chain.

13-1-7 Lock Management

Many operating systems only provide locks for entire files. Setting a lock means that all other updates or even all operations to the file are locked out. This can cripple system response time. A single lock per file does simplify the referencing mechanism. For every open file one lock is defined. When smaller granules are to be locked, the number of required locks is unpredictable, but often large.

Since shared databases require many locks, the management of the locks frequently becomes a responsibility of the database or the transaction management system. The semaphores used for locking have to be dynamically assigned to data items since it is not feasible to preassign a semaphore for every possible data item and its hierarchy. Locking schemes for databases, if implemented with care, can support the number of locks associated with a fine granularity.

A straightforward implementation to handle semaphores dynamically will use a linked list for the locked items. The identification of an item being locked and its temporary owner is entered into a list of locked items. When the lock is released the entry is removed from the list. On each reference which might conflict according to Tables 13-2 or 13-4 the list is checked to find if the object is locked. The search for locks requires a linear scan. Considerable CPU time will be used if many locks are active and have to be checked.

A more effective way to organize semaphores for locks is by using a hash table. Only one bit is needed to indicate that an object is locked, so that a modestly sized hash table can accommodate many locks. To minimize the number of collisions we choose a generous hash table size, say

$$m_s \approx 2\,\#(\text{transactions}_{max}) \times \#(\text{number of locks a transaction may set})$$

A single additional semaphore is used to protect access to the hash table when setting or testing locks. The unique object identification is the key to be transformed into a bit address in the table, and the setting bit at that address indicates whether the object is `occupied` or `free`.

The important notion in using a hash table for the semaphores is that *collisions are not detected or resolved.* The collisions due to the use of hashing cause some false lock indications, but the rate of *false-locks* can be set very small by increasing m_s. A transaction is typically blocked without checking if the lock is true or false; the only check needed is that the `occupied` signal is not due to another lock request from the same transaction. The transaction progress thread defined in Sec. 11-4-2 contains the required data.

The rate of overall false lock indications will increase in busy times, but the time needed to deal with them remains constant, and this a considerable advantage over the list-processing scheme, which uses more time per check when the demand is already high. However, the probability of deadlock (presented in Sec. 13-2-3) increases at busy times, and if lexical ordering is used to avoid deadlock the key-to-address transformation should be of the sequence-maintaining type, as described in Sec. 3-5-1.

Using the same space to keep object names for collision resolution would greatly reduce m_s and increase the number of collisions. To regain the low number of collisions and have the capability to resolve them the hash table has to be made many times longer, for a 10-byte object name by a factor of 81, and this may cause paging, requiring another level of locking!

The blocking of transactions and resources will not only affect performance, but unfortunate interaction sequences can delay the system *forever*. These problems are the subject of the next section.

13-2 HIBERNATION AND DEADLOCK

The reader may have developed an uncomfortable feeling in the preceding section. Multiple transactions walk around a database, claiming ownership of objects and regions in order to block others from interfering with their intended activities. There is indeed a distinct possibility of trouble. The problems can be categorized into two classes:

Hibernation A transaction is blocked, or put to sleep, and the system is too busy to rouse the transaction within a reasonable time: *the transaction is in hibernation.*

Deadlock One transaction is blocked by another, and the other transaction is blocked in turn by the first one. The circle can involve multiple transactions, and none of them can be awakened without potential for integrity violations: *the transactions are mutually deadlocked.*

From the user's point of view the difference between hibernation and deadlock is small: *the system doesn't work again.* For a system designer the difference is major: deadlock can be understood and solved or its occurrence may be reduced to tolerable levels. Hibernation problems can be due to one-time activities, and by the time an analysis is done all transactions are awake and no evidence remains. Deadlock also has been more interesting to the computer scientist because it is more amenable to formal analysis, whereas analysis of hibernation requires showing that interference can lead to excessive delays relative to some performance specification.

Hibernation was already encountered in Table 13-2. The classic deadlock condition is associated with incremental claims; Fig. 13-4 illustrates the problem.

13-2-1 Hibernation

Hibernation occurs when a transaction does not receive resources for an exccessively long period. The user who submitted the transaction which is in hibernation has reason to believe that the system is not functioning.

Hibernation of a transaction H can occur because there is an apparently infinite chain of waiting transactions preceding H. Thhis can be caused because another transaction holding a claimed resource of H does not terminate, because H has a low scheduling priority relative to heavily used other transactions, or because the blocking priorities do not permit H to proceed.

An infinite chain of updates can be created if a circularity occurs in the transactions or in the database constraints. If the `sales_manager` gets 5% of the `department.profit` as a `salary` but the `salaries` are deducted from the `profit` then we have a case which will execute until round-off errors, a transaction manager, or the MTBF intercede. Such a transaction is in error, but the error will not be detected if it was tested on `employee`s who do not share in the profit.

Having transactions of differing priorities is obviously an invitation for hibernation of transactions which have low priority. Processes which have heavy file demands may be assigned a high priority to the CPU so that they will make acceptable progress. This rule also leads to an improved disk utilization. An excessive number of such processes can however starve all other users. A technique to deal with starvation due to priorities, if priorities cannot be avoided, is to increase the priority of a process by some amount periodically, so that eventually the delayed process will have an adequate priority to proceed. Transaction systems tend to avoid priority schemes.

An example of the third case, hibernation due to blocking, can be caused by the basic locking rules shown in Table 13-2. The addition of the `if flag` clause provides a solution.

The lock set by an `Audit-read` blocks subsequent interfering `Updates`, but not subsequent `Audit-reads`. Since there are no restrictions on multiple `Audit-reads` being processed, an `Update` can be permanently delayed if interfering `Audit-reads` are submitted continuously.

The solution shown in Table 13-2 uses an additional `flag` semaphore. The setting of the `flag` to `on` delays entry of further `Audit-read`s and enables access for waiting `Update` requests. Alternative means to assure access for waiting `Update`s and avoid hibernation use priorities or preemption.

Long-term blocking can still occur when one transaction issues updates at a high rate. Now all audit-reads will be delayed. It is hence necessary that the system not only considers integrity; when there are delays the system has also the task to allocate the use of resources equitably among the transactions.

A solution to the allocation problem [Dijkstra^{65}] uses a semaphore to control a system variable which gives access to the object to each transaction in turn. An approach used in operating systems for page allocation may also be applicable: transactions which do not get their fair share of computing cycles receive an increased priority standing. Eventually the blocked transaction will be chosen from the queue.

A practical solution used by many transaction-oriented systems is to limit the length of time that a transaction is allowed to be active. This approach brings two new problems to the fore: The amount of time needed is influenced by system load, and a strict rule may cut off some valid transactions when the system is busy and let erroneous transactions proceed when the system is lightly loaded. This variablility can be controlled by more detailed accounting of CPU time; unfortunately, on many systems it takes a significant fraction of a CPU to account precisely for its use by multiple users.

The second, and more serious problem with killing a transaction which exceeds its time limit is: what does the system do within a transaction which did not finish? The fact that a lock was requested is an indication that an inconsistency could be expected. The only satisfactory solution is to attempt to undo the transaction. The logging and recovery facilities described in Sec. 11-4 are required to achieve this. The conditions for rollback will be further discussed in Sec. 13-2-4.

In systems which support extensive and short transactions the time limit is set according to time estimates specified by the programmer. This estimate may also be used for scheduling purposes. A long transaction can cause intolerably long blocking.

13-2-2 Causes of Deadlocks

Deadlocks occur when two or more transactions request resources incrementally and mutually block each other from completion. The transactions and resources form a cycle, seen in Fig. 13-4 and abstracted in Example 13-4. We will now also look at some other cases of deadlock and then treat the deadlock issue in general.

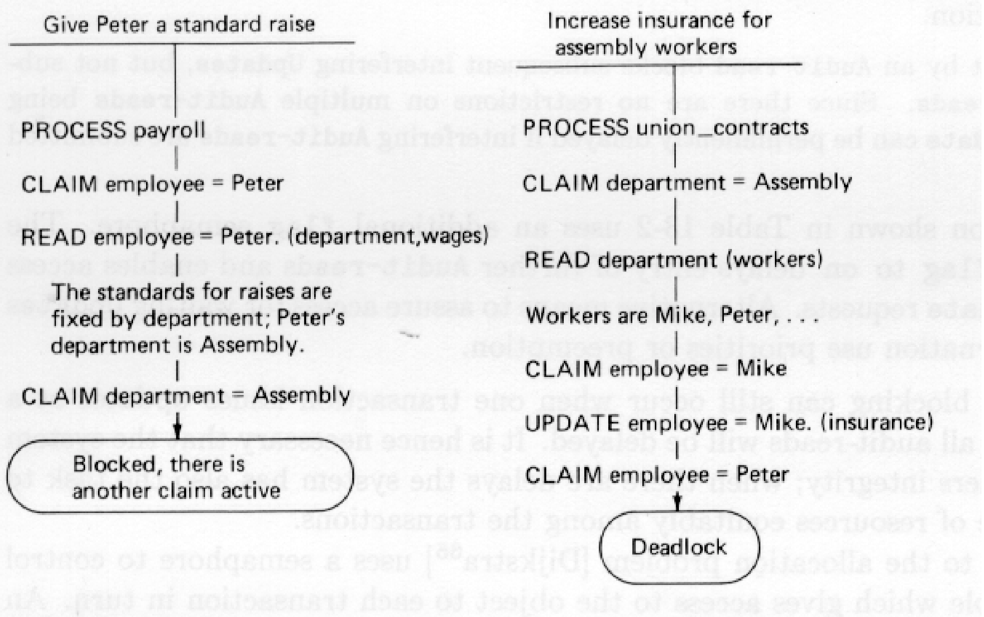
Give Peter a standard raise
PROCESS payroll
CLAIM employee = Peter
READ employee = Peter. (department,wages)
The standards for raises are fixed by department; Peter's department is Assembly.
CLAIM department = Assembly
Blocked, there is another claim active
Increase insurance for assembly workers
PROCESS union_contracts
CLAIM department = Assembly
READ department (workers)
Workers are Mike, Peter, . . .
CLAIM employee = Mike
UPDATE employee = Mike. (insurance)
CLAIM employee = Peter
Deadlock

Figure 13-4 Circular deadlock.

A deadlock can even be created when a single object is accessed by two transactions and incremental claims are accepted. An example is shown in Fig. 13-5.

This type of deadlock has a higher probability of occurring if the objects being locked are large. When a file is the unit of locking, the first claim may be issued to gain access to one record and the incremental claim issued in order to update another record in the file. Both claims address the same unit.

Example 13-4 Deadlock Cycle

Transaction A		Object		Transaction B
payroll	→	employee		
"	→	department	←	union_contract
		employee	←	"

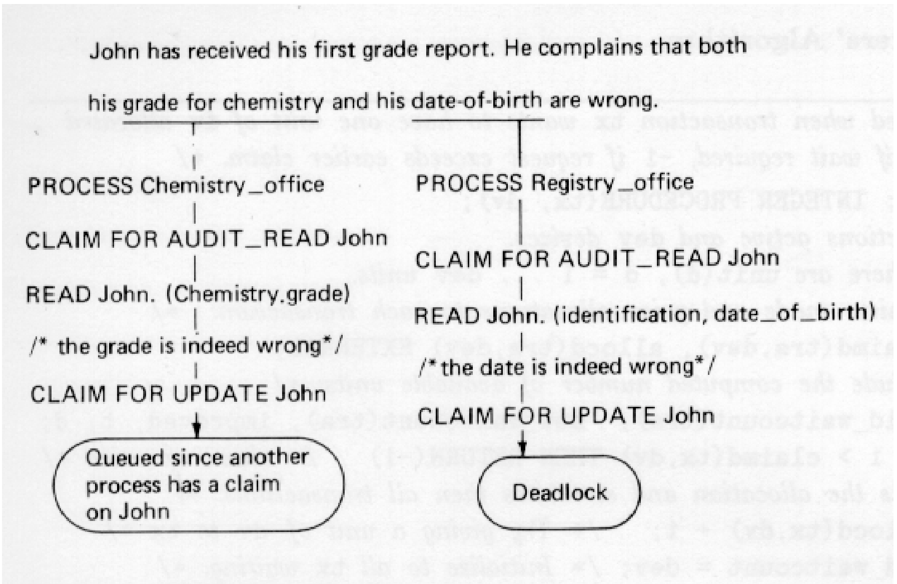
John has received his first grade report. He complains that both
his grade for chemistry and his date-of-birth are wrong.
PROCESS Chemistry_office
CLAIM FOR AUDIT_READ John
READ John. (Chemistry.grade)
/* the grade is indeed wrong*/
CLAIM FOR UPDATE John
Queued since another process has a claim on John
PROCESS Registry_office
CLAIM FOR AUDIT_READ John
READ John. (identification, date_of_birth)
/* the date is indeed wrong*/
CLAIM FOR UPDATE John
Deadlock

Figure 13-5 Deadlock on one object.

Deadlock Due to System Resource Sharing Deadlock can also be caused by competition for objects which are not specifically identified, but are members of a shared *resource class*. Temporary blocking occurs when resource limits are reached. A simple example is a system which has four tape units available to be allocated among the transactions. Let the claims of two transactions (P1,P2) for tapes be (3,3), and the initial allocation A1 be (2,1) as shown below. The sequence A2 leads to trouble, in the sequence A3 the blocking is only temporary.

Example 13-5 Deadlock in Resource Allocation

Transaction	Claim	A1	A2.1	A2.2	A3.1	A3.2	Limit
P1	3	2	2	$\rightarrow \cancel{3}$ Blocked	$\rightarrow$3	Can Complete $\rightarrow$ 0	} 4
P2	3	1	$\rightarrow$2	$\rightarrow \cancel{3}$ Dead	1	Blocked by P1	

At point A1 it is not safe to allocate (step A2) the remaining tape unit (#4) to transaction P2. If #4 were given to P2, a subsequent request for an allocation by P1, permitted by P1's claim, will be blocked (A2.2); P1 will not release its resources. Another request by P2 will cause deadlock since neither transaction can complete. Allocation step A3 is safe because it lets P1 complete, and then P2 can proceed. Other transactions with more limited claims, say one tape, may be scheduled for unused but claimed resources subject to the same constraints.

For the general case of resource sharing we have to consider multiple devices since deadlock can be caused by one transaction being blocked for, say, buffers from a finite buffer pool, while the other transaction is holding tape units. The algorithm to test for this occurrence is called the *bankers' algorithm* because it is similar to a simple view of commercial loan allocations. A program to decide if an incremental allocation can be safely made is presented in Example 13-6. The test is performed by iteratively simulating all transactions to eventual completion. Lack of success means the claimed unit should not now be allocated. In most successful cases only one iteration will be needed, one iteration of the algorithm requires on the order cases of #(transactions) · 2 #(number of device types) steps. This cost is still high.

Example 13-6 Bankers' Algorithm

```
/* This procedure is called when transaction tx wants to have one unit of dv allocated
   It returns 1 if ok, 0 if wait required, -1 if request exceeds earlier
claim. */
Allocate_claimed_unit: INTEGER PROCEDURE(tx, dv);
/* There are tra transactions active and dev devices.
   Of each device type there are unit(d), d = 1 ... dev units.
   Given are also the claims made and prior allocations to each transaction:  */
DECLARE unit(dev), claimd(tra,dev), allocd(tra,dev) EXTERNAL;
/* Working variables include the computed number of available units: */
DECLARE avail(dev), old_waitcount(tra),  new_waitcount(tra), improved, t, d;
   IF allocd(tx,dv) + 1 ¿ claimd(tx,dv) THEN RETURN(-1)   /* claim exceeded */
/* The procedure attempts the allocation and simulates then all transactions. */
   allocd(tx,dv) = allocd(tx,dv) + 1;   /* Try giving a unit of dv to tx */
   improved = 1;   old_waitcount = dev; /* Initialize to all tx waiting. */
   DO d = 1 TO dev;                             /* Add allocations to all */
      avail(d) = unit(d) - SUM(allocd(*,d));     /* transactions (*)  */
   END;                                       /* to compute avail.  */
   DO WHILE ( improved );                   /* Simulate completion of all tras */
      improved = 0;                       /* as long as resources are freed.  */
      new_waitcount = 0;                    /* Set computed number of blockages. */
      DO t = 1 TO tra;
         IF old_waitcount(t) ≠ 0 THEN  /* See if there are enough resources */
            DO d = 1 TO dev;            /* so this transaction   */
               IF avail(d) ≤ claimd(t,d) - allocd(t,d) THEN
                           new_waitcount(t) = new_waitcount(t) + 1;
            END;
         IF new_waitcount(t) = 0 THEN   /* can complete and eventually  */
            DO d = 1 TO dev;            /* return its allocated units.  */
               avail(d) = avail(d) + allocd(d);
               improved = 1;            /* It did.     */
            END;
      END;
      IF SUM(new_waitcount) = 0 THEN RETURN(1) /* Go ahead, all can complete!*/
      old_waitcount = new_waitcount;           /* Try if the new avail helps */
   END;                               /* by iterating again. */
   allocd(tx,dv) = allocd(tx,dv) - 1;   /* We can't do it, take unit back and */
   RETURN(0);                          /* Indicate that tx has to wait. */
END Allocate_claimed_unit;
```

Transactions which communicate in order to jointly process data are also subject to deadlock. An example of such cooperating transactions occurs when data are moved between files and buffers by one transaction and data are processed simultaneously by other transactions. A two-transaction example to accomplish double buffering was shown in Fig. 2-14. One transaction may generate data to be written to files at a high rate and acquire all buffers. Now a data-analysis transaction cannot proceed because the data-moving transaction cannot deliver additional data because of buffer scarcity. Since the data-analysis transaction will not give up its buffers until completion, there is again a classical deadlock situation.

Another problem to be addressed in communicating transactions is to avoid message loss when the arrival of a new message coincides with the clearing of an earlier message. This problem is solved using semaphores to introduce blocking of coincident arrivals and is not a primary database concern. The use of locks to resolve problems due to resource sharing does interact, however, with the management of locks in database systems.

Deadlocks Regions across Structural Levels Deadlocks can be caused by interfering demands on objects from distinct resources classes:

Transaction P1 needs a tape drive to write results from its disk region.

Transaction P2 cannot release a tape drive until it is no longer blocked from accessing a record in this same disk region.

Mechanisms to handle deadlocks hence must involve *all* lockable objects. This may include tape units, printers, and other devices, buffers and memory areas allocated by the operating system, and files, blocks, and records controlled by the file and the database system. This can cause problems in otherwise well structured systems, where devices are controlled on one system level and data objects are managed at higher system levels.

An unsatisfying solution is the use of potentially harmful global variables to communicate locking information. Strict rules regarding access privileges to these variables, and adequate documentation can avoid most problems.

Deadlock Conditions The potential for deadlock exists when all four conditions are true [Coffman71]:

1: Locks Access interference is resolved by setting and obeying locks.

2: Blocking An owner of an object is blocked when requesting a locked object.

3: Completion Guarantee Objects cannot be removed from their owners.

4: Circularity A circular request sequence, as shown in Example 13-4, exists.

All techniques to deal with deadlock attempt to change one of these conditions. Techniques which resolve deadlocks, however, will also affect hibernation. Any technique adopted should be analyzed to gain an understanding of its effect on system performance due to increased lock out and individual task response time due to blocking.

Frequency of Deadlock Occurrence Deadlock frequency will depend on the degree of interaction among transactions. When regions being locked consist of large and many objects, say multiple files are locked in various combinations, then deadlock will be frequent and will cause serious problems unless constraints are applied. Resolution involves isolation and duplicated files and is hence not well adapted to a database-oriented approach. In a very busy system deadlocks may yet be frequent where incremental claims specify small objects. A large system (WEYCOS), which has the capability to detect and recover from deadlock, experiences 100 deadlocks per hour [Bachman73].

Other published values range to as low as five [BrinchHansen73] or two per year [Frailey73] in systems which are programmed for limited deadlock prevention.

Statistics of system failure due to deadlock are difficult to gather in an environment that fails to consider their existence. As shared systems increase in popularity and activity it will become more difficult to ignore deadlock.

13-2-3 Deadlock Avoidance

Prevention is the better part of valor. An ability to avoid deadlock can simplify many alternative choices. Deadlock avoidance schemes impose, however, restrictions on users which can be difficult to accept.

Given the list of four conditions allowing deadlock to occur, there are four approaches to avoid deadlock. A fifth approach avoids both blocking and circularity.

1: Postrepair Do not use locks and fix inconsistency failures later.

2: Dont Block Only advise requesters of conflicting claims.

3: Preempt Remove objects from their owners if there is a conflict.

4: Presequence Do not allow circular request sequences.

5: Two-phase locking Make all claims first and if none are blocked begin all changes.

Postrepair The first approach, repair problems due to not locking afterward, can be valid in experimental and educational systems, but is unacceptable in most commercial applications. If the systems has already some means to repair damage because of errors, the approach may be tolerable.

An airline, for instance, permits 15% overbooking on its routes. If lack of locking contributes, say, 1% cases of overbooking, reducing the planned rate to 14% and avoiding locks can greatly improve performance. Recording how a customer paid for the flight still requires locking payment files but here no or little interference is expected.

Dont block The second approach puts the responsibility on the transaction. The system will give a notification of potential interference by denying the request for exclusive access. Access is still permitted.

The transaction may proceed, taking the risk that data will be modified or it may decide to restart later. If the transaction wants to be sure then it will *preclaim* all needed resources before accessing them. Since this approach also avoids circularity we will discuss it below. If the transaction goes ahead it may simply warn the submitter of the fact that interference has been detected. The transaction cannot count on the validity of its previous claims. This choice may have to be made when systems do not have restoration or backup capabilities.

Whether the database is ever left in an inconsistent state depends on the programming algorithms and the types of interference encountered. A KEEP statement of the 1973 version of the COBOL proposals (CODASYL[78,···]) will interrupt transactions which access records modified by currently active transactions. The competing transactions have the same choice; go on or quit.

If auditable results are required, no interference can be tolerated; the transaction may decide to release its previous claims and restart itself after some delay or it may abort entirely and let an external restart mechanism take over.

Preempt The third approach, preemption of claims granted to transactions, requires a rollback capability. Either the transaction, when it is notified that it cannot proceed, has to restore the database and place itself in the queue for another turn, or the system has to kill the transaction, restore the database, and restart the transaction anew. Since this approach depends on deadlock detection, it will be further discussed in Sec 13-2-4.

Presequence The fourth choice is to avoid circularity in request sequences. There are three approaches here, monitoring for circularity, sequencing of objects to avoid circularity, and two-phase locking, treated, because of its importance, in a distinct subsection.

To avoid deadlock the request pattern of all the transactions can be monitored. If conditions of potential circularity which can lead to deadlock are detected the candidate transactions can be blocked. The *bankers' algorithm* of Example 3-6 provided a similar example.

If the request cannot be satisfied because of the potential for deadlock, the request is queued. As claims are released, the transactions waiting in the queue are checked to see if they can be restarted. A scheduling algorithm can be used to choose among multiple candidates for processing, but at least one request should be granted to prevent scheduler-caused deadlocks [Holt^{72}]. In dynamic algorithms the transaction arrival order may have to be maintained.

Unfortunately the computation becomes quite complex when there are many resources, as is the case when we deal with data elements instead of devices. A proper analysis also has to consider serializability and compatibility of interfering operations. The computation takes much longer when many transaction are active; an undesirable feature during busy times. References to such algorithms are found in the background section.

‡‡‡‡‡‡ A simple algorithm, used in IBM OS, allows incremental claims only in ascending lexical sequence for the file name. The objects being locked are files. A proper pair of sequences to avoid deadlock is shown in Example 13-7.

Two ascending sequences cannot create a circular sequence so that the deadlock problem seems solved. This technique greatly limits flexibility for incremental requests. The IMS system operating under OS uses its own locking scheme, shown in Table 13-1.

Example 13-7 Deadlock Avoidance by Lexical Ordering

Transaction P1	Transaction P2
File AA	File BA
File BA	File BB
File BC	File BC
File BD	File BM
File BZ	File BZ
File CX	

‡‡‡‡‡‡

The ascending request rule can be relaxed by allowing requests to files lower in the lexical sequence to be made; but if such a request is denied the transaction is obliged to release all objects required up to this point and start over fresh. This extension can be seen as combining approaches 3 and 4.

Two-phase Locking A simple approach to avoid circularity is to require preclaiming of all objects before granting any locks. Claiming resources before promising to grant access to them means that a transaction may not be able to complete the preclaiming phase. Once all claims have been made locks can be set to protect the transaction from interference by others. This technique is called *two-phase locking* and works well in an environment using a two-phase commit protocol for transactions.

Two-phase locking is hence characterized by a phase where resources are acquired and a phase where they are used and released. A failure to acquire resources in phase one can cause the transaction to be aborted without much pain. No resources may be acquired during phase two, since a failure to obtain them would require undoing of changes made to committed resources.

The major problem with two-phase locking is that preclaiming can lead to having to claim more and larger objects than will actually be needed. If a computation on part of the data determines what further object is needed, an entire file rather than a record may have to be preclaimed.

In Example 1-1 we determined from a `Department` file the name of the `manager` who was to receive a raise and found `"Joe"`. Preclaiming requires locking of the entire `Employee` file since we do not yet know which `Employee` will be updated. Not preclaiming and finding the `Employee.name = "Joe"` record claimed could mean that an interfering transaction is just demoting `Joe` to a position which does not qualify him for the raise.

In order to reduce the granule size prereading may be used here, as presented in Sec. 13-1-2. The notification mechanism used in non-blocking schemes can warn the transaction of interference potential.

The simpler rule is that a transaction shall claim the entire region to be locked at one, and make no incremental claims. System primitives are used to lock out claims by other transactions until the availability of all objects in the region is verified and the entire claim is recorded. To issue a comprehensive claim requires perfect foresight; the cost of failure is having to release all acquired resources in their original state, and initiating a new claim. A transaction, in order to reduce its probability of failure, will claim a generous region.

Summary Deadlock avoidance methods suggest a familar theme, *binding* (Sec. 1-6). By imposing restrictions on the choices of the objects to be accessed, multioperation is made safe. This in turn allows a major improvement in the productivity of the system.

As stated earlier, the system may not allocate the claimed objects to the user until they are actually needed. Prior to allocating an object which is the subject of another claim, a check will be made if now a deadlock potential is created. Deferred allocation of claims is used in Burroughs 6500 systems.

Reserved Resources Deadlock caused by competition for classified resources (tape units, disk drives, memory) is sometimes mitigated by not allowing any new transactions to be started when the utilization reaches a certain level. This technique will not assure deadlock avoidance unless the reserve is kept impractically large, enough to allow all active transactions to complete. This technique is used, for instance, by

IBM CICS in its buffer management. Operational experience can be used to balance the amount of reserve against the deadlock frequency to achieve optimum productivity. The gathering of sufficient deadlock statistics for throughput optimization is an unpleasant task in systems which do not provide rollback.

13-2-4 Deadlock Detection

Deadlock detection is by definition too late. The only cure when deadlock is detected is to kill one transaction as gracefully as possible. Deadlock detection is, however, an important task in the maintenance of system integrity.

If deadlock avoidance is already part of the system, then deadlock detection provides a backstop which allows the correction of faults in programs or hardware. In an environment where the responsibility for deadlock avoidance is placed on the programmers, a deadlock detection facility is vital. Anyone who has ever attempted to debug a running on-line system which contains initially a few deadlocked transactions, but which slowly becomes paralyzed as another transactions attempt to access locked objects can recall the horror of working against time, knowing that at some point the debugging transaction itself will be blocked. The solution, canceling all transactions and restarting the system, leaves one waiting for the next occurrence. There are systems in operation which live with the expectation of occasional deadlocks, owing to the desire to optimize performance.

In systems which cannot legislate incremental claims out of existence, deadlock is unavoidable and has to be detected and, when detected, resolved. Deadlock detection algorithms are similar to deadlock avoidance algorithms. A circular chain of dependencies has to be found. In Fig. 13-6 transactions P1, P2, P4 create the circular deadlock (c, g, k, j, h, f, e, d).

However, transactions P3 and P5 are also blocked until the deadlock is removed. The deadlock can be broken by application of approach 3: *remove owned objects from a transaction.* The least number of objects in the circle are owned by

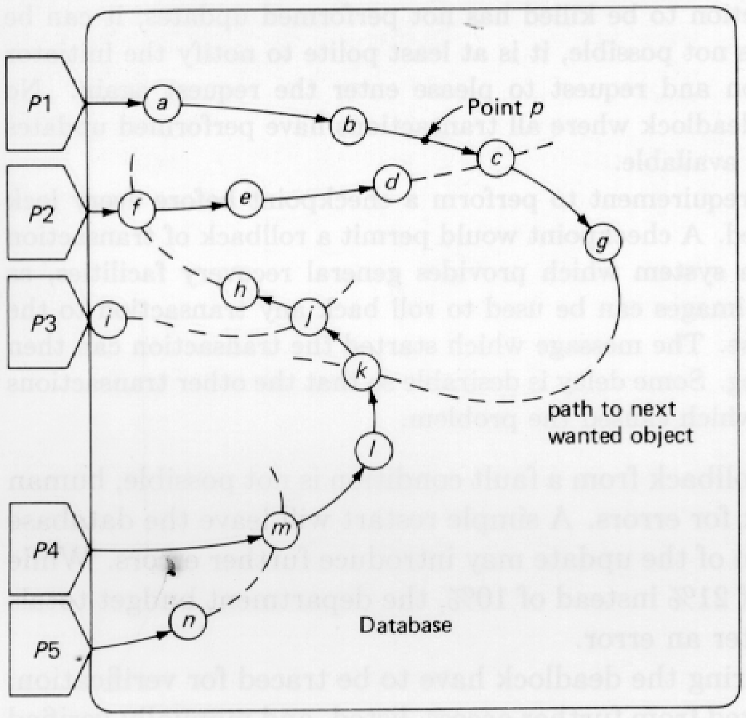

Figure 13-6 Deadlock and blocking.

transaction P1 (c, g), but it is in general, difficult to restart a transaction at an arbitrary point (p) unless a checkpoint was made. In a transaction-oriented system with recovery facilities an entire transaction can be restarted, so that transaction P2 is the candidate for assassination; it owns the least number of objects (f, e, d). Only objects which have actually been modified should be counted when the transaction to be killed is selected.

Deadlock detection by logical analysis is impossible if the desired object cannot be identified. A transaction which waits for a request from any of a number of terminals is an example. The transaction sleeps until awakened, and then it will check the reason for the alarm and do something appropriate. Figure 6-10 showed a file system process which can respond to user service requests, file operation completed signals, and time-exceeded signals. It is desirable that these processes do not themselves claim sharable resources so they can remain outside of the domain of deadlock management.

Time limits placed on transaction execution times provide another means to detect deadlock, blocking, or hibernation. The guilty party cannot be positively identified when a time limit is exceeded. In the example above, it is possible that transaction P5 exceeds its time allotment first. Killing P5, however, will not resolve the deadlock condition.

The action to be taken when a deadlock is detected will depend on the available recovery facilities. One popular multiuser system stops. This allows console diagnosis of the problem but leaves remote users quite unhappy.

Preemption If the transaction to be killed has not performed updates, it can be abandoned. If automatic restart is not possible, it is at least polite to notify the initiator of the transaction of the situation and request to please enter the request again. No satisfactory solution to resolve a deadlock where all transactions have performed updates exists unless rollback facilities are available.

If updates are infrequent, a requirement to perform a checkpoint before every lock involving an update can be imposed. A checkpoint would permit a rollback of transaction P1 to point p in Fig. 13-6. In a system which provides general recovery facilities, as described in Sec. 11-4, the before-images can be used to roll back any transaction to the point where it entered the database. The message which started the transaction can then be replayed from the transaction log. Some delay is desirable so that the other transactions have a chance to pass the object which caused the problem.

Correction If automatic rollback from a fault condition is not possible, human intervention is required to check for errors. A simple restart will leave the database inconsistent, and the reexecution of the update may introduce further errors. While Peter might appreciate a raise of 21% instead of 10%, the department budget totals may will not reflect this fact after an error.

All transactions blocked during the deadlock have to be traced for verification. Affected records have to be blocked from further access, listed, and manually verified and corrected. If blocking is not done, any inconsistencies introduced can further contaminate the database. A correction may require real world data, i.e., *"How many widgets do we really have in stock"* after an inventory file failure.

Where databases support financial data, the accountant's tradition of double entry bookkeeping can be helpful. Since any transfer of funds from one account to another is accomplished by matching pairs of debit and credit entries, manual correction of a damaged database is simplified. When the accounting entries are further properly identified with the name of the originator and the number of the transaction, a deadlock recovery system can present the request for correction in a simple form. Automatic correction (without rollback) may be possible but is not advisable since no adequate audit of data entry exists when transactions are not properly completed.

13-2-5 Techniques

The effort to avoid or detect deadlock can be significant. The time is affected by the number of transactions and the number of objects being locked. If both are small, a (transaction·object) matrix can describe the locking pattern. The matrix elements need only one or two bits (allocated, or claimed and allocated). For claims to resource classes, integers to denote the number claimed and received will be needed. If there are many objects (locking units are small: blocks or records), a static matrix becomes infeasible and dynamic lists have to be managed.

The better algorithms to detect deadlock, for instance, the bankers' algorithm of Example 13-6, require on the order of (#transactions · #objects) operations. In addition, all claims, allocations, and releases have to be noted. If the probability of deadlock is low, it may be feasible to test for deadlock only periodically. This may allow the deadlock to spread as other transactions attempt to use the blocked regions. It is important that the deadlock checking algorithm itself does not need to claim resources which might be blocked.

Transaction-oriented systems tend to mimimize potential interference by keeping the number of active transactions small. A high priority is assigned to transactions which hold resources so that they will leave the system as fast as possible. Two lists are adequate for deadlock detection, one to keep track of objects and one to record the state of each transaction. For each active transaction there is an entry in the lexically ordered object list. If an object when requested is found already in the list, and the owning transaction is blocked, there is a possibility of deadlock and the test for circularity is made.

13-2-6 Deadlock Management in Distributed Databases

Deadlock involving multiple nodes of distributed system can occur when transactions lock resources at multiple nodes. A deadlock in a distributed system can be localized to a single node. Subtransactions created by a remote transaction can be involved in a local deadlock. Except for the fact that the cost of aborting a subtransaction can be hard to evaluate and may well be high, the methods for handling local deadlocks remain unchanged.

In this section we consider deadlocks involving multiple nodes. The four principles of deadlock creation and resolution apply also when the deadlock involves more than one node. The risk and the cost of deadlock can be greater because the communication times increase the interval that resources may have to be held. The notion that a remote user can claim and block local computations is also in conflict with the concept of distributed authority and responsibility.

Deadlock and Replication The replicated elements we find more frequently in distributed databases also increase the chance for deadlock.

A change-of-address notice has been sent to two sites A, B by a customer who keeps accounts at multiple branches of a bank. The bank maintains an integrated database model and considers the customer a single entity having replicated data at A and B. At both sites the change-of-address is entered, at site A a subtransaction which will also update B's copy is created, and at branch B a subtransaction for A is created. Each subtransaction finds its object busy. If the subtransactions lock their objects a deadlock is created.

We see all four conditions for deadlock (lock, block, hold, circularity) in this example. Not locking on a change-of-address is quite feasible, and the possible inconsistency from simultaneous updates may not harm much. If the date and time of any update is kept, a periodic validation can repair missed updates. In this example we used a data element whose consistency requirements can be relaxed.

The next scheme in terms of desirability addresses the fourth notion of deadlock prevention: avoidance of circularity. By designating one copy to be the primary copy, i.e., making only one node responsible for any one element, circularity in replicated copies is avoided. Here the rule might be that the nearest branch of the bank keeps the primary address. If a static assignment cannot be tolerated, a *primary-copy token scheme*, as presented in Sec. 11-5-4 will avoid deadlocks in replicated elements.

The remaining two strategies, avoidance of blocking or preemption, have a high cost in distributed systems and no special advantage in the replicated element case.

We considered in this example a data element which is not critical in terms of consistency, but the resolutions we found give a general direction.

Conflicts for many elements which have consistency constraints can be avoided by giving only one site update privileges. If this is not feasible, the primary copy token scheme will prevent deadlocks due to replicated data. These strategies address the second and fourth notions of deadlock prevention by having only one owner for a replicated element.

Deadlock Avoidance The general case of distributed deadlock is caused by transaction, which access several distinct objects. To avoid deadlock in the general sense in distributed systems again the avoidance of locking is most attractive. When locking can be avoided, first the frequency of deadlock is reduced and second the cost of deadlock resolution in the remaining cases becomes less. Having fewer elements participating in deadlock resolution reduces the communication and processing costs greatly.

A reasonable way to reduce locking in distributed systems is to restrict the claim privilege from remote nodes. A remote user may, for instance, not be given any right to claim local locks for read purposes. In that case the remote users can obtain data, perhaps with time and date stamps, and with an indication whether currently a local transaction had a claim on these data. Any decision on the validity of the data so obtained is made at the remote site, which can reissue the transaction if the value of the acquired data appears risky.

Preemption may be similarly modified by always giving priority to local transactions. This will reduce the effect of remote users on local transactions. The remote transaction has to reinitiate any canceled subtransaction. Since the remote transaction must in any case deal with communication failures, we can safely assume that a remote transaction has already the capability to restart its subtransactions.

The fourth solution, avoidance of circular request sequences, requires a global naming scheme. This scheme appears mainly of value in tightly controlled and strongly bound systems, since otherwise it is difficult to assume that all objects can be ordered, since the name to object correspondence may be a local option.

Distributed Deadlock Detection Distributed deadlock detection is initiated when a node finds that it is deadlocked. If the deadlock involves subtransactions submitted by remote nodes, a node has to collect information of all dependencies in the network, as shown for a local net in Fig. 13-6. The information needed can be always sent along when a subtransaction is submitted, or has to be obtained when a problem is detected by querying the other nodes. To resolve the deadlock a transaction involved in the chain has to be preempted. The selection of transaction can be based again on minimal loss, or on features of certain sites. Some sites may have good rollback capability or be deemed unimportant. If the information is available to all nodes, all deadlocked nodes can come to the same conclusion and resolve the deadlock identically. Queries for the information to resolve a deadlock after the deadlock has occurred must be answerable without access to any data which may be blocked.

These strategies are liable to fail if there is another system failure. The simple local priority scheme leading to preemption and avoiding deadlocks caused by remote transactions seems to be more robust.

13-3 MAINTENANCE OF INTEGRITY

The previous sections have shown how integrity can be violated, and how such violations can be avoided or their effect restored. These discussions are of necessity based on expectations of perfect hardware and software performance. Well managed system design and programming techniques can considerably improve system reliability.

A large multiuser on-line file system [Frey71] developed under management by the author operated for more than 6 years without loss of data stored on files. Control over reliability, security, and integrity was achieved by integrating access, access protection, and backup. The granule for all operations was the block; fields could be protected only by encryption (Sec. 14-3). Periodic monitoring was performed to locate errors. Backup procedures provided manually initiated rollback.

Continued integrity of a database is essential for successful operation. We would like to have assurance that the mechanisms used to protect shared data cannot fail. In practical systems today correctness of integrity protection is impossible to prove. Many levels of hardware and software have to work without failure to permit integrity protection mechanisms to do their work.

We covered in earlier Chaps. 11 and 12 the prerequisite reliability and access protection issues. Secs. 13-1 and 13-2 presented the locking mechanisms used to avoid problems due to shared access. We will now present some further considerations which can help designing and operating a database so that integrity is kept at a high level.

13-3-1 Programming Integrity

Most systems allow the programmers to make decisions regarding the use of locks. Problems may be caused by having one individual make decisions which affect programs written by others. A system organization to prevent update interference can be achieved if:

1 The *access protection system* allows only one user, the owner of the data, to modify a protected object or to set a lock for the object.

2 The *access security system* allows only one instance of the user to exist.

3 The *file management system* locks objects which are synonymous with the objects identified for ownership by the access protection system.

The last condition is rarely true. Ownership is allocated to logical entities: fields, records, files, databases, whereas physical control is exercised over physical objects: blocks, tracks, files, and devices. If interference problems can be resolved only on the common file level, shared database access is severely restricted. The rules outlined above do not solve read interference if query regions involve files of more than one user.

If the programmer is not restricted by the access system from potential interference with the activities of others, a higher level of control may be required. Update operations may be permitted only if a claim has preceded the operation. This removes the decision *to lock or not to lock* from the purview of the programmer, although it does not guarantee the locks will cover the correct region. Some system automatically precede update operations with a claim which will lock the object being updated, but an additional claim facility is required to let the programmer claim regions containing multiple objects. For the sake of programming consistency explicit claims are desirable.

The decision whether a read operation is of the *audit* or *free* type is local to a transaction and program. It is desirable to let the end user know which choice was made when the results are being transmitted.

The general problem of assuring the integrity of databases has not been solved. An initial step is the definition of the application requirements by collecting semantic constraints.

13-3-2 Integrity Monitoring

Since a single consistency failure can be gradually copied throughout the database, a strategy of regular monitoring of the database is essential wherever long term data are kept. If results based on data stored for many years are in error, the user's confidence in all the work of past years is suddenly lost.

Monitoring is possibly on two levels: structural and content-oriented. *Structural monitoring* can be carried out by the file system without user participation. *Content-oriented monitoring* requires the user to provide assertions regarding data relationships. In either case monitoring is possible only if there is some redundancy in a database.

Structural Monitoring Structural monitoring is often conveniently combined with periodic dumping operations. One of the areas to be verified is storage allocation: all blocks in the domain of the system should be owned by only one user or belong to the free space list. Some systems may also have a pseudo-owner for bad blocks – either physically damaged storage or blocks containing data which may be in error or are backup for data which may be in error.

Where pointers exist, their logic can be verified: rings should form cycles, trees should not form cycles. Indexes should point only to their primary file. Where data are ordered the sequences can be verified. An entry in a key sequence which is out of order can make successor records unavailable or can make a binary search fail.

In Sec. 9-5 we indicated the desirability to avoid *essential links*. Redundant symbolic references can be used both for integrity monitoring as well as for restoration of the database. Symbolic references are less volatile and can be used to reset pointers if an object is found which corresponds to the reference argument.

Content Monitoring Some data-checking processes can be driven using information from the schema: high and low limits and code validity. The TOD system provides for the inclusion of checking database procedures, to be executed during database verification runs. This allows greater cross checking of database files than is possible at data entry time, and is applied to all database entries to assure equal integrity of old data. Data content summaries, presented graphically can be helpful to users who have a certain expectation regarding data.

Assertion statements, now being introduced for program verification, can also be used to verify the database. Possible assertions are

Example 13-8 Integrity Constraints

```
Department.salaries = SUM(Department.employee_salaries)
Apprentice.age ¡ 30
COUNT(Children) ≤ 12
Manager.salary ¿ Employee.salary — Employee.manager=Manager.name
Employee.social_security_number ≠ UNDEFINED
```

Example 9-11 showed such statements from INGRES. Assertions which are too restrictive will cause excessive inconsistency reports, so that some tuning may be required.

Since data conditions are apt to change frequently, it is desirable that a content monitoring process can be table-driven by these statements. If continuous reprogramming is required, frustration will soon lead to disuse of integrity monitoring.

Summary The maintenance of database integrity is a major problem in any database operation. Unless the correct data are reliably available management cannot be expected to depend on database systems. If manual backup systems are kept in operation because of distrust of the computer operation, few economic benefits can be obtained. In databases which are intended for public benefits an unreliable operation will reap scorn and derision. Promises that the next version of the system will be better will be evaluated in the light of the promises made previously.

BACKGROUND AND REFERENCES

Integrity in a system has to be provided at many levels. The initial concerns about maintenance of integrity appeared in communication systems which had to manage multi-node message traffic, and later in operating systems which had to allocate resources to multiple users. Many of the initial applications were such that simple locks could solve the problem without causing deadlock. Only as locks became safe and common did deadlocks become an issue. We have described some problems in databases which require locks. Careful reading of the literature is often required to see which lock problem is being discussed. The principles are similar, but the validity of any method will depend on the application. Everest in Klimbie^{75} distinguishes the lock types required in a database environment. Since locking is closely related to protection many references from Chapter 11, for instance Kohler^{81}, are also relevant.

We have assumed that secure lock primitives are available within the operating system. A seminal paper by Dijkstra^{65} defines the tools and their application. The algorithms for processes which share data, presented by Dijkstra, were further developed by Knuth^{66}, deBruijn^{67}, Eisenberg^{72}, and Lamport^{74} (*the baker's algorithm*). Hellerman^{75} illustrates the problems. Dykstra also initiated the *banker's algorithm* for allocation of claimed resources; this was worked out in 1969 by Haberman^{76} and improved by Holt^{72}, applying graph theory to the analysis.

BrinchHansen^{73} and Bayer^{78} consider many aspects of the management of locks in operating systems. Dennis^{66} describes a locking mechanism. Coffman^{71}, Collmeyer^{71}, and develop the requirements for locking in concurrent database access. Locking in B-trees is developed in Bayer^{77}. Gray in Kerr^{75} and in Bayer^{78} develops locking rules for regions of various granularity which combine claim and allocation protection. Interaction tables are provided for various file organization methods. Ries^{77} looks at the performance tradeoff. Hawley^{75}, Everest in Klimbie^{75}, and Engles in Nijssen^{76} provide more examples and the rules for CODASYL. Olle in Douque^{76} reviews them critically. Locking and transactions are defined by Eswaran^{76}. Mechanisms for a distributed system are described by Rosenkrantz^{78} and in Bernstein^{80S} and related papers.

Denning^{71} provides a fine description of deadlock and Lamport^{78} covers timing of events in distributed systems.

Mathematical and graphical techniques have been extensively developed to understand integrity problems in computer systems. Holt^{72} and Genrich^{73} present graphical approaches to deadlock algorithm development. Algorithms to locate cycles in graphs are found in Aho^{74}. A survey for distributed systems is provided by Bernstein^{81} and the performance of the protocols is analyzed by GarciaMolina^{81}. An excellent overview of the basic mechanisms is given by Bernstein^{82}.

Dijkstra^{71} develops in a very careful manner the algorithm (*the dining philosophers*) to allocate shareable resources. Deadlock can be prevented because the claim pattern is known beforehand. The byzantine generals algorithm (Pease^{80} and Lynch and Dolev in Wiederhold^{82}) addresses the problem of voting (Thomas^{79}, GarciaMolina^{82E}) in the presence of faulty processors. Presser^{75} summarizes deadlock problems and solutions.

A system which immediately allocates all claims is described by Reiter^{72}. Havender^{68} describes the allocation for resources held throughout successive computations in IBM OS. Frailey^{73} developed a sparse matrix technique to avoid deadlock in Purdue's MACE operating system for the CDC 6500. Pirkola in Arden^{75} uses a binary matrix to protect files in Michigan's MTS.

The concept of rollback used to be distasteful to many computer scientists (Yannakakis^{82}) since it implies failure of prevention algorithms. Papadimitriou^{79} defines serializability and Lynch^{81} and Fischer in Aho^{82} address the constraints imposed by it.

Bachman[73], reporting on WEYCOS (Fichten[72]), finds a major productivity benefit obtained from rollback of transactions versus greatly restricted access in a large database operation. These optimistic algorithms have been formalized and analyzed by Kung[81] and Schlageter[81].

Fossum in Klimbie[75] describes the facilities for deadlock management in UNIVAC DMS 1100. Chamberlin[75] describes the locking mechanism of SEQUEL. Deadlock detection is analyzed by Menasce[80] and Obermarck[81].

The distinction between audit-reads and free-reads (GarciaMolina[82W]) is not always explicitly stated in the literature, so that comparison of algorithms can be difficult. Audit-reads can be always be performed in systems which use versioning as shown by Adiba[80] and Fischer[82] .

How timing problems lead to constraint violations is discussed by Morey[82]. Frey[71] describes several structural verification techniques for a file system. The (TOD) system by Wieder-
hold[75] includes schema directives for database audit. Buneman[79] defines *alerters* to monitor the database. Eswaran in Kerr[75], Gray in Neuhold[76], and Chamberlin[76] specify an integrity subsystem. In Andler[82] multiple processors are employed. Hammer in Kerr[75] and Stonebraker in King[75] apply integrity assertions at data entry time, but the concept is, of course, extendable to integrity monitoring. Lafue[82] considers the tradeoff. Hammer[78] and Bernstein[80B] improve the efficiency of integrity monitoring schemes.

EXERCISES

1 Determine the difference in access cost for the two update procedures shown in Fig. 13-3. Indecate where locks should be placed to make the left choice safe. If setting a lock is equivalent to a disk write and checking of a lock is equivalent to a disk read, which choice is preferable.

2 Draw a flowchart to avoid the deadlock illustrated in Fig. 13-4 using the preclaiming approach.

3 Program a deadlock detection scheme for Fig. 13-4. Estimate its running time given p processes and n objects.

4 a Devise some integrity assertion applicable to the sample problem you have been using since Exercise 1 of Chap. 1.

b Describe the size and quantity of the objects you use.