On Decoupling Concurrency Control from Recovery in Database Repositories

by

Heng Yu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2005

©Heng Yu, 2005

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

We report on initial research on the concurrency control issue of compiled database applications. Such applications have a repository style of architecture in which a collection of software modules operate on a common database in terms of a set of predefined transaction types, an architectural view that is useful for the deployment of database technology to embedded control programs. We focus on decoupling concurrency control from any functionality relating to recovery. Such decoupling facilitates the compile-time query optimization.

Because it is the possibility of transaction aborts for deadlock resolution that makes the recovery subsystem necessary, we choose the deadlock-free tree locking (TL) scheme for our purpose. With the knowledge of transaction workload, efficacious lock trees for runtime control can be determined at compile-time. We have designed compile-time algorithms to generate the lock tree and other relevant data structures, and runtime locking/unlocking algorithms based on such structures. We have further explored how to insert the lock steps into the transaction types at compile time.

To conduct our simulation experiments to evaluate the performance of TL, we have designed two workloads. The first one is from the OLTP benchmark TPC-C. The second is from the open-source operating system MINIX. Our experimental results show TL produces better throughput than the traditional two-phase locking (2PL) when the transactions are write-only; and for main-memory data, TL performs comparably to 2PL even in workloads with many reads.

Acknowledgements

I would like to express my heartfelt gratitude to Professor Grant Weddell, my supervisor, for his guidance and encouragement during my program. His careful supervision, insightful suggestions, and exceptional patience prove invaluable to the completion of the thesis.

I am also grateful to Professor Ken Salem and Professor David Toman, my thesis readers, for taking their precious time reviewing my thesis and for their constructive comments.

I give thanks to my fellow students in the database research group, Huizhu Liu, Lubmir Stanchev, Lei Chen, and Khuzaima Daudjee, for the discussions with them which give great benefit to my research. Special appreciation goes to Dr. Yijun Yu at the University of Toronto, whose critical suggestions helped me to publish some of my thesis work in CASCON 2004.

It is my luck to meet wonderful friends in the University of Waterloo, without whom my life as a graduate student would have been far less interesting. I would like to thank Ningyan Zhong and Yi Zheng for always treating me with delicious food and giving me a feeling of family. Thanks also go to Saul Warhaft, Ramesh Sathiyaa, Mihaela Gheorghiu, Claude-Guy Quimper, Kim Honeyford, and other friends in the Minota Hagey Residence for sharing with me the marvellous years there. I would also like to thank Leo Jingyu Lee and other brothers and sisters in the Chinese Christian Community, from whom I have received strong spiritual support.

I give my deepest thanks to my parents, Yingchuan Yu and Enduo Wang, for their forever love, support, and understanding.

The financial support from Nortel and NSERC for my research work is gratefully acknowledged.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Applying database technology to embedded control programs

Repository is a widely used software architecture in which the software modules or subsystems interact with one shared data source (Figure 1.1). The inception and maturation of relational database technology since the 1970s have had a great influence on this style of architecture. With a general purpose database engine installed, the component subsystems communicate with the database engine in the SQL language. This evolved repository architecture is called *database repository*, which is shown in Figure 1.2. The database repository architecture along with database technology give software developers considerable benefits, referred to as *DB payoff*. The payoff happens for two reasons. First, SQL provides the subsystems with a conceptual view of the database which in turn reduces their development cost. Second, each software subsystem itself automatically inherits from the database engine solutions to concurrency, reliability and security problems.

A new application area to apply the database repository architecture is *embedded control programs*. An embedded control program is a software system consisting of a collection of subsystems that interact with common data in main memory through a set of predefined transaction types [48]. Therefore, an embedded control program naturally has the repository architecture. There are many examples of embedded control programs in the

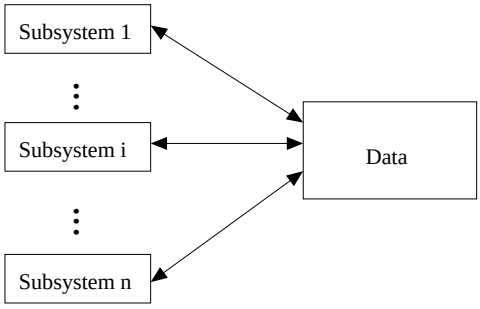


Figure 1.1: Repository architecture

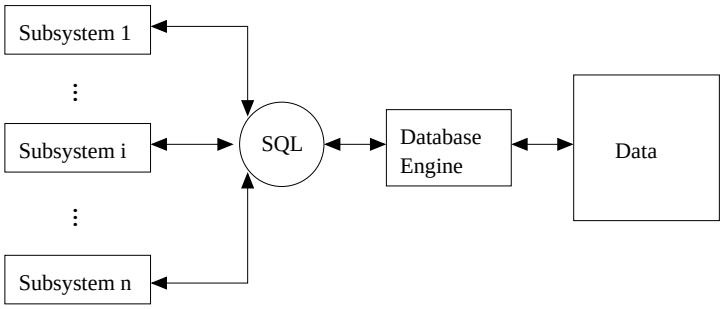


Figure 1.2: Database repository architecture

IT industry. Many legacy systems, *e.g.*, control programs of private branch exchanges (PBXs) and operating systems, belong to this category. The introduction of inexpensive handheld computers in wireless networks and mobile computing environments initiates a new field of embedded control programs. If we adopt database repository architecture for embedded control programs and supply them with appropriate database management, embedded control program software development and maintenance will also benefit from the DB payoff. The application of database technology in embedded control programs has double contributions. First, it realizes the information integration of legacy data with a uniform SQL/OQL interface. Second, it supplies database facilities for the new lightweight applications on wireless devices.

Such applications have strict resource limits and high throughput requirements. Traditional relational database engines are not suitable for this area. In particular, it is intolerably costly to process queries in an interpretative way at run time. Because all transaction types are predefined, query plans and the code that implements the plans can be compiled at system generation time [48]. Because compile-time processing plays an important role in the database management of embedded control programs, we call the embedded control programs that are equipped with such database techniques *compiled database applications* [44]. In the thesis, the terms "embedded control program" and "compiled database application" are used interchangeably.

1.2 Concurrency control for compiled database applications

The thesis addresses the issue of *concurrency control* of compiled database applications. As in traditional database systems, we use the classical *transaction model* [8] [23] [53]. The transaction pattern is "Begin Transaction/Commit/Abort Transaction", which allows users to commit or abort transactions explicitly. The well established standards of correctness include *serializability*, *deadlock-freedom*, and *recoverability*. In general-purposed database systems, such standards are ensured by the following approaches. The *two-phase-locking* (*2PL*) [18] is applied to guarantee serializability. Deadlocks are detected by timeout or looking for cycles in a waits-for-graph, and resolved by aborting selected victim trans-

actions. Moreover, the system keeps an on-disk log for all updates for potential recovery from aborts.

We use the transaction model for compiled database applications as well. Specific to such applications, we find recovery from arbitrary transaction aborts will introduce very complicated logic, which makes the generation of query processing code difficult. In addition, we conjecture that logging all updates to the common database to support very general abort capabilities may result in performance degradation. Therefore, it critical to decouple concurrency control from recovery in compiled database applications.

In traditional DBMSs, there are two reasons why general recovery mechanisms are required.

(a) They may be needed to support the transaction model itself if it is possible for clients to explicitly issue an abort.

(b) Since deadlock is possible under 2PL, there is an internal need to abort transactions.

To avoid the above two situations, naturally two things are necessary. First, we shall adopt a much simpler "Begin Transaction/Commit" transaction model for the compiled database application domain. Second, we need to replace 2PL with a locking protocol that guarantees both serializability and deadlock-freedom.

The first goal is not difficult to obtain for compiled database applications. Because the transaction types are predefined, and the conditions and behaviors of transaction abort are predictable, we can implement a high-level "abort" by "committing" compensations [20][1]. So it is feasible to remove the "Abort Transaction" termination from our transaction model.

For the second goal, a locking protocol that guarantees both serializability and deadlock-freedom is *tree locking* (*TL*) [42]. We give a broad survey of the research that has been done on TL and its variants in Section 2.2. An approach to apply such a protocol efficiently in the compile database application context is the key part of the thesis work.

[1]To illustrate this, we give a case study of the Linux system call **fork** processing in Appendix A.1.

1.3 Thesis overview and organization

In the thesis, we model predefined transaction types as finite state machines, and characterize the workload with probabilities of transaction types. Based on the model, we have designed runtime algorithms to lock and unlock data items following the TL protocol. For compile-time processing, we have given a heuristic method to build lock trees and other relating data structures from the knowledge of transaction types and workload, and explored inserting locking operations into the transaction types to save the runtime overhead.

Two workloads have been built up for our experiment. The first is derived from the OLTP benchmark TPC-C [2], which contains a set of predefined transactions that access shared data tables concurrently and thus resembles the workload of a compiled database application. The second is from the source code of selected system calls in the open source operating system MINIX [46], in which we try to reflect more features for memory-main data processing. We conducted our experiments on the two workloads to evaluate the performance of TL in comparison to the traditional 2PL.

The organization of the remainder of the thesis is as follows. Chapter 2 surveys the related work, including other work on compiled database applications (Section 2.1), TL and its variants (Section 2.2), and non-flat transaction models (Section 2.3). Chapter 3 fully covers our application of TL to compiled database applications in order to ensure both serializability and deadlock-freedom. Section 3.1 defines the transaction model, then Section 3.2 introduces our runtime locking and unlocking algorithms following the TL protocol, and Section 3.3 presents the compile-time processing that generates the lock tree and inserts the lock steps into the transaction type. Chapter 4 presents how we built two workloads from TPC-C (Section 4.1) and MINIX system calls (Section 4.2) respectively. Chapter 5 shows our experimental results under the two workloads. Chapter 6 concludes the thesis and proposes future work.

Most of the research work in Chapter 3 and the corresponding experiments in Chapter 5 have been published as a conference paper [58]. With slight differences in the experimental part, they are also available as an earlier technical report [57]. The workload part in Chapter 4 is available as another technical report [59].

Chapter 2

Related work

2.1 Compiled database applications

In collaboration with Nortel Ltd., the DE**MO** (Design Environment for Memory-resident Object-oriented database) project [3] [4] [48] investigates how to employ database technology to aid the software design and maintenance of embedded control programs. Besides the concurrency control issue addressed to in the thesis, the other two directions of research in the field are compile-time *query processing* and *physical design*.

2.1.1 Query processing

[48] introduces the topic on accessing embedded control data via a database query interface at the conceptual level. The legacy data structures must be retained at the physical level, and the queries should be complied into C or Java programs that manipulate physical data with performance comparable to the code handwritten by programmers. The relationships between the conceptual schema and the physical level layout are captured by *integrity constraints* and *binding patterns*. The compile-time query processing includes *query expansion*, *plan generation*, and *code generation*. The paper gives a practical algorithm for the resource-bound plan generation. An overview of the query processor architecture is also presented.

[13] uses a *existential graph* reasoner [39] to generate query plans from the OQL queries.

The goal is designing a compile-time query optimizer which generates efficient low-level code under the main memory limitation. The *schema information*, including *class definition*, *constraints*, *indexes*, and *encoding*, is defined in the *Universal Data Representation* (*UDR*), and encoded by an existential graph. Moreover, both queries and access plans are also represented in existential graphs. [13] also defines the *plan algebra* for the output plan, which can be easily transformed to code in programming languages. First, a query is transformed to a query graph using the existential graph reasoning service *expand* over the encoded database schema. Then a plan graph is built based on the query graph and schema graph. The plan graph represents a cheap plan that can answer the query. Finally, from the plan graph, a plan in the form of the plan algebra is generated. The query processing in [13] corresponds to the query expansion and plan generation phases in [48]. And the final code generation phase [48] from plan to programs is trivial.

In [34], *description logic* ($\mathcal{DL}$) reasoning [10] is applied for query optimization. The paper defines an object query language. The control data structures are encoded in a $\mathcal{DL}$ language $\mathcal{DLFDE}$ that supports path functional dependencies [51] and equalities [50]. The reasoning of concept subsumption for stratified $\mathcal{DLFDE}$ terminologies is decidable in exponential time, which is acceptable for compile-time query processing. A set of query transformation rules based on the $\mathcal{DL}$ reasoner are provided. With such rules, the original query can be expanded to an equivalent query over the low-level physical structures. The generated query can be easily transformed to a plan that scans indexes in nested loops.

2.1.2 Physical design

[31] provides an extension to the *Universal Data Representation Interchange Language* (*UDRIL*) to capture the data structures at physical level. The extension allows declaration of a store model in terms of *object identities*, *stored structures*, *arrays* with their *sizes*, and *inlined data structures* with their *offsets*. It enables an incremental physical design that adds assertions to the conceptual database schema to represent the concrete data encoding. [31] shows that the obtained UDRIL sublanguage has the power to express the types in C language. The algorithms that transform between UDRIL and C structures are given.

[44] and [45] address the physical design of compiled database applications. [44] considers *exogenous indexes* (the indexes that the searching structures and the actual data are

separated, *e.g.*, B+-tree) while [45] considers *endogenous indexes* (those that the actual data are included in the searching structures, like B-tree). The data model used is the object model with functional dependencies [51], and in particular represented in the $\mathcal{DLFD}$ language (a $\mathcal{DL}$ language that has a order dependency [22] construct) in [45]. The workload consists of *critical queries*, *updates*, and *non-critical queries*. It is required the physical design should consume small storage and support efficient processing of critical queries and updates. The approach in [44] and [45] is transforming queries into *parameterized access requirement type* (*PART*), then merging the PARTs to reduce storage, and finally building physical design from the merged PARTs. Moreover, [45] proves that selecting the smallest number of indexes that can be queried and updated efficiently is intractable. Therefore, it is probably difficult to further improve the exponential-time index selection algorithm given in [45].

2.2 Tree- and DAG-based locking protocols

In retrospect, the widely used *2PL* is proposed in [18]. The paper shows that 2PL is both necessary and sufficient for serializability *if the transaction accesses an arbitrary data set*, or *even if the data set is unknown beforehand*. In practice, strict 2PL protocol, which releases all of a transaction's locks at the time it commits or aborts, is typically used. However, 2PL has two disadvantages. First, 2PL forces a transaction to hold locks on data items which it no longer accesses until no further locks are requested by it. This reduces concurrency. Second, 2PL is not deadlock-free. To alleviate the first problem, variants of 2PL are proposed. For example, *altruistic locking* (*AL*) [40] allows a transaction to donate data items it locks but no longer uses so that other transactions can access them. The second problem can not be solved inside the 2PL protocol itself, but can be handled by additional deadlock detection techniques, *e.g.*, timeout or waits-for-graph [8]. As we mentioned in Section 1.2, the subsequent aborts for deadlock resolution are very unsuitable for compiled database applications. We are more interested in deadlock-free locking protocols. The remainder of the section surveys TL and its variants.

2.2.1 Basic TL

The basic *TL* protocol is originally proposed in [26] and [42]. It takes advantage of the information of data and transactions to allow early releasing of locks while still guaranteeing the correctness of transactions. Assuming there is a lock tree structure whose nodes are the data items accessed by the transactions, and all operations are writes, the basic TL rules are:

TL Rule 1 A transaction can operate on a data item only after it obtains a lock on it.

TL Rule 2 Except for the first data item that a transaction locks, a data item can be locked by a transaction only if the transaction is currently holding a lock on its parent in the tree.

TL Rule 3 After a transaction release the lock on a data item, it can never get the lock on it again.

It is proved in [42] that tree-locked schedules are both serializable and deadlock-free.

2.2.2 DAG locking

The basic TL can be generalized from tree structures to directed acyclic graph (DAG) structures. [26] considers two types of locking conditions for DAGs, corresponding to TL Rule 2 for trees. They are *weak condition* and *strong condition*. The former allows a transaction to lock a node if it is holding lock on any parent in the graph, and the latter allows one to lock a node only if it is holding locks on the majority (more than half) of its parents in the same biconnected component[1]. [26] concludes that tree structures (either "point from the root" or "point to the root") with the weak condition guarantee serializability and deadlock-freedom, and that DAG structures with the strong condition guarantee serializability and deadlock-freedom. Therefore, we have a serializable and deadlock-free DAG locking protocol, named as the *strong DAG locking protocol*, which is similar to TL except it replaces Rule 2 by:

[1] According to [43], a *biconnected component* of a graph consists of a maximal collection of nodes u_1, u_2, $\cdots$, u_p such that either $p \geq 3$ and for each pair (u_i, u_j) there exists two or more disjoint chains between u_i and u_j, or $p = 2$ and u_1 and u_2 share an arc.

- Except for the first data item that a transaction locks, a data item can be locked by a transaction only if the transaction currently holds locks on the *majority* of its parents in the same biconnected component of the graph.

Another version of DAG locking, the *rooted DAG protocol*, is proposed in [54]. It requires that the DAG has a single source, and the corresponding locking condition of a node is:

- Except for the first data item that a transaction locks, a data item can be locked by a transaction only if the transaction is currently holding locks on some of its parents and has locked all its parents (some may be already unlocked) in the graph.

The rooted DAG protocol also guarantees serializability and deadlock-freedom.

2.2.3 Guarded locking

In [43], the TL protocol, the strong DAG protocol, and the rooted DAG protocol, are generalized to a *guarded protocol* that is also based on DAGs. The generalized protocol takes connectivity of DAGs into account. A DAG $G = \langle V, E \rangle$ is a *guarded graph* if and only if with each node $v \in V$ there is a set of pairs:

$$guard(v) = \{\langle A_1^v, B_1^v \rangle, \cdots, \langle A_{n_v}^v, B_{n_v}^v \rangle\}$$

that satisfies the conditions:

1. $\emptyset \neq B_i^v \subseteq A_i^v \subseteq V$;
2. the nodes of $\bigcup A_i^v$ are parents of v;
3. if $A_i^v \cap B_j^v = \emptyset$, then there is no biconnected component of G including nodes from both A_i and B_j.

Example 2.2.1. *For the DAG in Figure 2.1 (from [27] with some modifications), the guards of its nodes are:*

$guard(v_1) = guard(v_2) = guard(v_3) = guard(v_4) = \emptyset$
$guard(v_5) = \{\langle \{v_1, v_2\}, \{v_1, v_2\} \rangle, \langle \{v_1, v_3\}, \{v_1, v_3\} \rangle, \langle \{v_2, v_3\}, \{v_2, v_3\} \rangle, \langle \{v_4\}, \{v_4\} \rangle\}$
$guard(v_6) = \{\langle \{v_4\}, \{v_4\} \rangle\}$
$guard(v_7) = \{\langle \{v_5, v_6\}, \{v_5\} \rangle, \langle \{v_5, v_6\}, \{v_6\} \rangle\}$

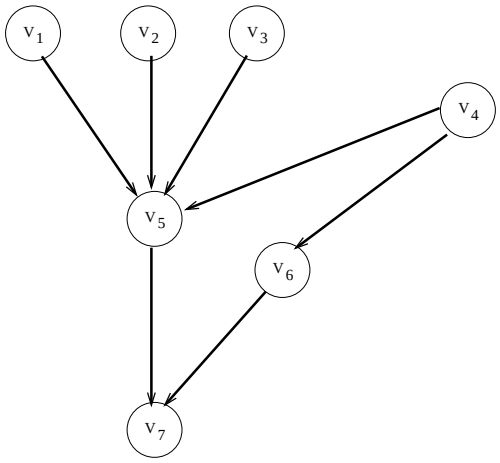


Figure 2.1: A Guarded Graph

The rules of the guarded protocol are:

1. A transaction can lock any node first, but to lock any subsequent node v, it must be holding a lock on the nodes in some B_i^v and must have locked the nodes in $A_i^v - B_i^v$.

2. A transaction may unlock a data item at any time, but it can lock it only once.

The generalized guarded locking protocol guarantees both serializability and freedom from deadlock. As a special case, the TL protocol has $guard(v) = \{\langle\{v\text{'s parent}\}, \{v\text{'s parent}\}\rangle\}$.

2.2.4 Tree/DAG locking with shared locks

The non-2PL locking protocols surveyed so far allow only exclusive locks. [27] extends TL and the generalized guarded locking to allow shared locks. Simply adding shared locks into the original protocols may violate serializability. There are two approaches with additional restrictions. The first one allows only transactions that contain either all writes or all reads, and requires write-only transactions to lock the root at first. This protocol, called *heterogeneous protocol*[2], ensures both serializability and deadlock-freedom.

[2]Although the two protocols are proposed in [27], the name "heterogeneous protocol" and "homogeneous protocol" are given in [36].

The second one is a *homogeneous protocol*, which extends guarded locking protocol and allows a transaction to have mixed read and write operations. It singles out *pitfalls* in a DAG w.r.t. a transaction. Each pitfall is a subgraph spanned by the nodes read by the transaction plus their neighboring nodes written by it. The homogeneous protocol additionally requires transactions to lock its *pitfalls* in the DAG following 2PL. However, the homogeneous protocol is no longer deadlock-free.

[36] furthers the protocols by allowing lock conversion from a shared lock to an exclusive lock (upgrade) and vice versa (downgrade). It is easy to add lock conversion into 2PL protocol by treating upgrading similar to locking and downgrading as unlocking, and by restricting upgrading in the growing phase and downgrading in the shrinking phase. In the same way, the two approaches that allows shared locks in [27] can be also extended to allow lock conversion. Both the new heterogeneous protocol and the new homogeneous protocol (named as *MGLP' protocol* and *M-pitfall protocol* in [36] respectively) guarantee serializability but neither is deadlock-free. It is shown that in MGLP' deadlock detection and elimination are cheap and incur no rollback. And it is also possible to modify MGLP' protocol to obtain deadlock-freedom by adding a new lock mode *update*. An update lock conflicts with exclusive locks and update locks but is compatible with shared locks, and only update lock (not shared lock) can be upgraded to exclusive lock. Deadlock handling in M-pitfall protocol is more difficult.

The homogeneous protocol in [27] cannot avoid deadlock, and may introduce costly cascading rollback at the time of deadlock resolution. For this problem, [25] proposes the *superpitfall* protocol, which is a restriction of the original homogeneous protocol. For each pitfall, the locking phase in which its root nodes are locked is defined as the *initial phase* (*IP*), while the locking phase in which other nodes are locked is defined as the *final phase* (*FP*). The superpitfall protocol restricts the homogeneous protocol by disallowing any unlocking in the IP and FP stages of each pitfall. Although it is not deadlock-free, it guarantees that no cascading rollback is necessary.

[28] proposes the *queue protocol* (*QP*), which is based on the M-pitfall protocol. The paper lists several problems in TL and M-pitfall:

1. In TL, locking a node has dual functionality: accessing the current node and accessing its successors. This causes low concurrency.

2. M-pitfall is not deadlock-free.

3. M-pitfall reduces the concurrency because of the two-phase locking requirement on each pitfall. This is called *pitfall paradox*.

To break the duality in the first problem, [28] redefines the operations as:

- ***LS***, ***LX***, and ***LU*** lock a node in shared, exclusive, and update mode, respectively. Meanwhile, they also acquire the right to lock the successors of the node.
- ***SKIP*** does not lock the current node, but acquires the right to lock its successors.
- ***UP*** upgrades the lock on a node from shared mode to exclusive mode, ***DN*** downgrades it, and ***UN*** releases the lock. Different from the basic TL protocol, a transaction which executes **DN** on a node does not lose the right to access its successors.
- ***QUIT*** gives up the right to lock successors.

Each node is associated with two FIFO queues [28]:

access queue the waiting list of transactions requesting lock to the node;

proceed queue the waiting list of transactions requesting to lock the successors of the node.

Therefore, the operations are the above operations are implemented as:

- **LS**, **LX**, and **LU** put the transaction at the tail of both queues. When the operation comes to the header of the access queue, the object is locked in the corresponding mode.
- **SKIP** puts the transaction at the tail of the proceed queue. **UP**, **DN**, **UN** do to the lock upgrade, downgrade, and release on the node respectively.
- **QUIT** deletes the transaction from the proceed queue.

Based on the above concepts and implementation, [28] defines the rules of QP as:

1. The first node locked by a transaction T can be arbitrary.

2. Subsequently, T can lock or upgrade a node n by **LS**, **LX**, and **LU** only if there is a guard $\langle A_i, B_i$ of n (See Section 2.2.3) such that T is currently at the header of the proceed queues of all the nodes in B_i and has been at the header of the proceed queues of all the nodes in $A_i - B_i$.

3. T must two-phase locking each node.

[28] proves that QP is both serializable and deadlock-free. Moreover, the second rule reduces the scope of two-phase locking from pitfalls to individual nodes. Therefore, the pitfall paradox is disposed of.

2.2.5 Hypergraph locking

Serializability

In [55] the serializability (safety) is studied for generalized transaction locking policies (protocols), including 2PL, TL, and the DAG locking. It assumes only write operations and exclusive locks. The model of transaction, locking, and safety are defined formally. The paper shows that 2PL is the only serializable policy that satisfies both:

1. it does not lock any data that the transaction does not operate on, and

2. it works for transactions with all possible sequences of data operations.

[55] also studies the conditions of serializability for a set of locked transactions and shows that decision of whether a set of locked transactions is non-serializable is an NP-complete problem. So serializability of a locking policy cannot be tested efficiently in general. To address this problem, the paper studies a subset of locking policies, *L-policy*. A locking policy P is an L-policy if P can be described by a set of conditions that state whether a given entity can be locked at a certain moment in a transaction, depending on the prefix of the transaction up to this moment. L-policy is a natural restriction of locking policies, and almost all well-known protocols, *e.g.*, 2PL, TL, and the DAG locking, belong to this category. The paper proves that the serializability of transactions locked by an L-policy can be tested in polynomial time. To model L-policy, [55] proposes the *hypergraph policy* (*HP*).

As defined in the paper, *directed hypergraph* $DH = \langle V, E \rangle$ is a hypergraph, in which each hyperedge $e \in E$ is a set of nodes in V. One node in e is specified as its *head*, and the remaining nodes in e are its *tail*. The underlying graph of DH is $H = \langle V, E \rangle$ without the head-tail specification on the hyperedges. A *path* in H from node x_0 to x_{n+1} if there is a sequence $x_0 e_0 x_1 \cdots x_n e_n x_{n+1}$ where $e_i \in E$ and $x_i \in V$ and $x_i, x_{i+1} \in e_i$ for $i = 0, \cdots, n$. A set of nodes S *separates* node x and y if deleting all hyperedges that contain nodes in S removes all paths from x to y. HP assumes that the data items are organized in a directed hypergraph DH, and the rules of HP are:

1. The first locked node can be arbitrary.
2. A subsequent node x can be locked if and only if
   (a) there is a hyperedge e of DH with head x, whose tail is currently locked or have been locked up to this moment;
   (b) for each y previously unlocked, the set of nodes currently locked separates x from y;
   (c) each node can be locked by a transaction only once.

[55] shows that an L-policy is serializable if and only if it is covered by the hypergraph policy for some directed hypergraph DH. 2PL is a special case of HP with DH as a clique, while TL is one with DH as a tree.

Freedom from deadlock

Based on [55], [56] further investigates freedom from deadlock of serializable locking policies. The paper first studies the case of two transactions and shows a locking policy with two transactions is serializable and deadlock-free if and only if it is contained in the multiple source DAG policy[3] for some DAG, which can be tested efficiently. However, testing

[3]Multiple source DAG policy (MSDP) [56] relaxes the restriction of a single source in the rooted DAG protocol [54]. It allows arbitrary first lock, but changes the condition to lock a subsequent node n to

- the transaction has previously locked all parents of n that are not separated by n from the first node of the transaction, and

whether a transaction set (with more than two transactions) under a locking protocol is deadlock-free is intractable, and even testing whether a serializable L-policy is deadlock-free is NP-complete, which is harder than testing serializability in [55]. An interesting feature that the paper [56] finds is that whether a serializable L-policy is deadlock-free depends only on the order in which data items are locked, and has nothing to do with when they are unlocked. Therefore, if one serializable L-policy always lock the same items in the same order as another serializable L-policy, either both or none of them are deadlock-free.

[56] also explores the sufficient conditions for freedom from deadlock. The paper gives the following definitions. A graph $D(P)$ is defined with regard to a locking protocol P. The nodes of $D(P)$ are data items, and there is an arc $\langle y, x\rangle$ if there is a legal transaction under P that starts by locking y and accesses x. Let $R_T(L_x)$ be the set of data items referenced by transaction T before it locks x. And let $F_1(x)$ be the set of parents y of x, for which there is a transaction T that starts with y, accesses x, and in $D(P)$ there is no ancestor z of x in $R_T(L_x) - y$. Define $D'(P)$ as a subgraph of $D(P)$ such that an arc $\langle y, x\rangle$ is in $D'(P)$ if there is a transaction T under P starting with y and accessing x, and $F_1(x) \cap [R_T(L_x) - y] = \emptyset$. The paper shows, for a locking protocol P which is both serializable and deadlock-free:

1. $D(P)$ is acyclic.

2. If y is an ancestor of x in $D(P)$, then in all transactions that contain both x and y, y gets locked before x.

3. If x is accessed by transaction T but is not the first entity T locks, then T references at least some parent y of x in $D'(P)$ before it locks x.

Based on $D'(P)$, some sufficient conditions of deadlock-freedom of a serializable protocol P are presented in the paper:

- If P is a serializable L-policy and $D'(P)$ is a tree, then P is deadlock-free.

- the transaction holds a lock on at least one parent of n.

- If P is a serializable and deadlock-free policy, and $D'(P)$ is a tree, then any prefix of schedules under P policy (assuming all locks held are released at the end of the prefix) are also serializable and deadlock-free.

- For an L-policy, if $D'(P)$ is not a tree, there may be undirected cycles in $D'(P)$ as shown in Figure 2.2. Suppose P is a serializable L-policy and $D'(P)$ is acyclic and contains no undirected cycles in Figure 2.2 (b). Then P is deadlock-free if and only if it satisfies the condition: if y is an ancestor of x in $D(P)$, then in all transactions that contain both x and y, y is locked before x.

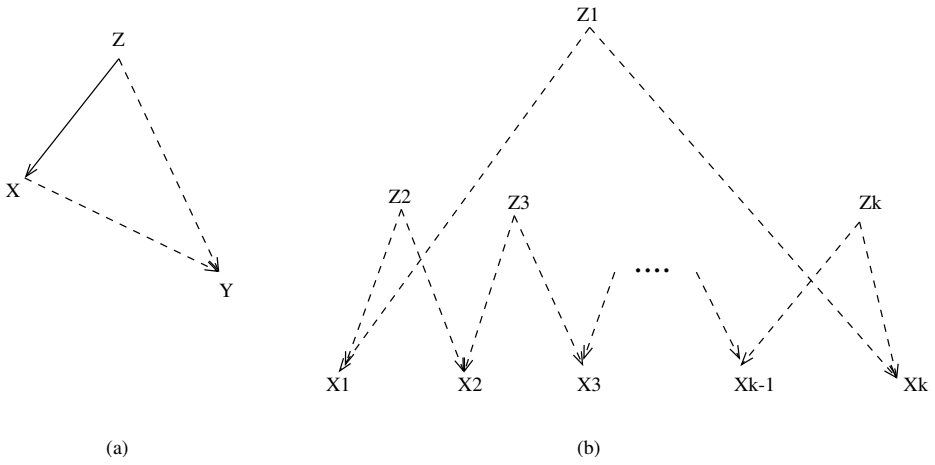


Figure 2.2: Undirected cycles in $D'(P)$ (From [56])

For the TL protocol, $D'(P)$ is the tree itself. So both the first and second conditions apply to TL.

2.2.6 Edge locking

[12] introduces the *edge locking* protocol to replace node locking in TL and DAG locking. It is noticed that in these protocols the exclusive lock on a node has multiple roles, *i.e.*, to protect the data itself and to prohibit access to the subgraph underneath the node. The paper shows that the latter functionality is ill-conceived, and suggests using edge

locking to lock the subgraph. And original TL and DAG locking can be transformed to the corresponding edge locking, following the rules given. Example of write-only TL and read-write TL are presented to show that edge-locking may have better concurrency than the original node locking. In general, both L-policies [55] and L-policies extended with write-only and read-only transactions [27] can be transformed to their edge locking versions. It is shown that the edge locking protocol is serializable (deadlock-free) when its original version is serializable (deadlock-free).

2.2.7 Dynamic TL

[17] introduces a variant of the TL protocol, *dynamic TL*. It addresses the problem of potential concurrency loss in basic TL. The cause of the problem is that a transaction under the TL protocol usually locks more data items than it actually accesses, especially when the data items accessed are dispersed in the tree. So even transactions with disjoint access sets may have overlapping lock sets in the tree. It is very difficult to find a static tree in which the access set of each transaction is localized so as to relieve the problem. Therefore, [17] proposes using a dynamic tree. The idea is keeping a tree for each transaction. When a new transaction comes for execution, its access set is analyzed, and the trees of the existing transactions that share data with the new transaction are merged with the tree of the new transaction to build a new tree. The combined trees can also be decomposed to remove the influence of the already finished transactions. The decomposition can be done at *system quiescence state* when there is no locked transactions. Even when the system is busy with rare quiescence states, deletion of a node is safe if

1. it is under the *strong condition* (the node has been locked and unlocked by all executing transactions that may lock it), or

2. it is under the *weak condition* (the node is no longer access by any currently executing locked transactions) and the schedule is a set of dynamically *strongly tree-locked*[4]

[4][17] defines strongly tree-locked on several prior concepts. Let t be a locked transaction, and let α be a set of database objects. The *restrictions of t to α*, denoted as $t|_\alpha$, is the subsequence of t that results from the removal from t of all steps (access, lock, and unlock) referencing database objects not in α. For a database tree Δ and a set of database objects α, let $\Delta^{-\alpha}$ be the set of database trees $\{\Delta_1, \cdots, \Delta_n\}$ that

transactions in the context of node deletion.

It is proved that the dynamic TL is still serializable and deadlock-free [17].

[17] leaves the work of changing the tree to the transaction management system at the time when a transaction comes or exits. In contrast, [32] allows each transaction to change the lock tree structure by itself. The added operations that change the tree structure are ***switch*** (change the parent of a node), ***add_child***, and ***remove_child***. When all transactions are assumed to be write-only, **switch** operation can be done if the transaction holds locks on both the old and the new parents, **add_child** can be done if it holds lock on the node under which a child is to be added, and **remove_child** can be done if it holds locks on both the node and its child to remove. We can furthermore allow read operations, and the method is similar to the heterogeneous protocol in [27] to allow write-only transactions and read-only transactions. [32] shows that the new TL protocol with these tree restructuring operations added still guarantees serializability and freedom from deadlock.

2.2.8 Dynamic directed graph locking for dynamic databases

[15] studies the locking protocol for a dynamic database context, which has three kinds of mutual exclusive operations: ***access***, ***insert***, and ***delete***. A sequence of insertions and deletions determines a *state* of the database. A schedule must be *proper* in that all the operation steps of all the transactions must be *defined* in the database state they are executed, *i.e.*, in a state a transaction can **access**/**delete** an item only if it exists in the database, and can **insert** an item only if it does not exist. The paper proposes for this model the *dynamic directed graph* (*DDG*) locking protocol, whose locking structure is a directed graph derived from the database. As a data item is inserted or deleted in the database, the graph is updated and gets a new state accordingly. Furthermore, different

---

results from the deletion of nodes in α. Let $V(G)$ be the set of vertices over which the graph G is defined, and let t be a locked transaction which accesses data set $A(t)$, t is *strongly tree-locked w.r.t. a database* Δ if $t|_{V(\Delta_i)} \in TL_{\Delta_i}$ for each Δ_i in $\Delta^{-(V(\Delta)-A(t))}$. That is, if for each database tree Δ_i that results from deleting from Δ those vertices not in $A(t)$, the restriction of transaction t to the vertices of Δ_i is tree locked w.r.t. Δ_i.

from the DAG protocol in Section 2.2.2, the graph can have cycles. The locking rules are [15]:

DDG Rule 1 Before T performs any operation (**insert**, **delete** or **access**) on a node v (or an edge $\langle u, v\rangle$), T has to lock v (both u and v).

DDG Rule 2 A node that is being inserted can be locked at any time.

DDG Rule 3 Each node can be locked by T at most once.

DDG Rule 4 A transaction may begin by locking any strongly connected component (SCC)[5].

DDG Rule 5 And subsequently, all the nodes of an SCC are locked together in one step provided that all the entry points[6] of that SCC in the present state of G have been locked by T in the past and T is now holding a lock on at least one of them.

[15] shows the DDG protocol is both serializable and deadlock-free.

[14] extends the DDG protocol to the *DDG-SX* protocol that allows **read** operations and shared locks. The rules are similar to those of the DDG locking except the following changes:

***DDG-SX Rule* 1** Before T performs any **insert**, **delete** or **write** operation on a node v (or an edge $\langle u, v\rangle$), T has to lock v (both u and v) in exclusive mode. Before T performs a **read** operation on a node v (or an edge $\langle u, v\rangle$), T has to lock v (both u and v) in either shared or exclusive mode.

***DDG-SX Rule* 5** All nodes on an SCC are locked together in one step if:

(a) All the entry points of that SCC in the present state of G have been locked by T in the past and T is now holding a lock on at least one of them, and

[5] A *strongly connected component* G_i of a directed graph G is a maximal set of nodes such that for each pair $u, v \in G_i$, there is a path in from u to v. An SCC is non-trivial if it has more than one node.

[6] Let G_i be a strongly connected component of graph G. An *entry point* of G_i is a node w in G but out of G_i, such that there is an edge $\langle w, v\rangle$ in G and v is in G_i.

(b) For every node v on this SCC that is a child of an entry point, and every path $v_1, \cdots, v_p, v$, $p \geq 1$ in the present state of the underlying undirected graph of G, such that T has locked v_1 (in any mode), and $v_2, \cdots, v_p$ in shared mode, T has not unlocked any of $v_1 \cdots, v_p$.

[14] shows that the DDG-SX protocol is serializable but not deadlock-free. It is obvious that the DDG and DDG-SX protocols are natural extensions of DAG protocol [54] and its variant homogeneous protocol [27], respectively. They extend the unit of locking from a node to an SCC to deal with cycles in the locking graph. And in the DDG-SX protocol the condition DDG-SX Rule 5(b) corresponds to the requirement of 2PL on pitfalls in [27]. [14] also shows that both altruistic locking [40] and dynamic TL [17] still guarantee serializability in dynamic databases with insert, delete, and (exclusive) access operations.

2.2.9 Applications of TL in index searching

There has been much research on the currency control of tree-structured indexes, especially B-trees and B+-trees[5] [29] [33] [30] [37] [35] [41]. And these works are summarized in textbooks [8], [23], and [53]. Many techniques in this area are closely related to TL and DAG locking.

1. *Lock coupling* [8] [53] (*lock crabbing* [23]) is crucial for B-tree/B+-tree traversal. It requires a searching operation to first hold the lock on the parent node before it acquires the lock on the node itself during a top-down searching in a B-tree. This conforms to the TL protocol in [42].

2. When we have ***insert*** and ***key_search*** operations together, we consider them as transactions. Both **insert** and **key_search** traverse the B-tree from the root. The former may change a node and and its ancestors because of possible node splits, while the latter is read-only. Therefore, we can view the case as the heterogeneous TL protocol with write-only transactions and read-only transactions [27], and the locking is both serializable and deadlock-free. Because of the properties of B-trees, there is an additional requirement for **insert**. An **insert** transaction must hold exclusive locks on ancestor nodes along the path downwards until it finds a *non-full*

node. Then it knows the ancestors above the non-full node will not be split and can be unlocked safely [53].

To improve the concurrency, the technique can be modified to allow **insert** to have lock conversions, similar to the lock conversion for the heterogeneous TL protocol suggested in [36]. An **insert** transaction first locks the nodes along the path in *will-write* mode, and converts to write lock on the nodes it updates. Because will-write lock does not conflict with read lock, this reduces unnecessary conflicts between **insert** and **key_search** operations. The modified protocol is still serializable and deadlock-free [8].

3. When we consider ***range_search*** in addition to **insert** and **key_search** on a B+-tree structure, we can link the leafs in ascending order of the keys to facilitate **range_search**. A **range_search** first goes down from the root to a leaf, and then follows the leaf chain to search on the designated range. Therefore, we have a DAG instead of a tree. Following the result in [26], a naive thought is to satisfy the strong locking condition (locking the majority of parents) to lock a node, *i.e.*, to lock a leaf, each **insert**, **key_search**, or **range_search** has to hold a lock on the leaf's parent in the B+-tree and its predecessor in the leaf chain, which results in low concurrency. [53] shows this is unnecessary given the semantics of B+-tree operations, *e.g.*, both **key_search** and **range_search** are read-only, and the **insert** that causes leaf split only adds the new node to the right of the original node. Therefore, the weak condition (locking only one parent/predecessor) suffices. As a consequence, in the proposed technique, an **insert** can lock a leaf in exclusive mode if it holds an exclusive lock on its B+-tree parent; a **key_search** can lock a leaf in shared mode if it holds a shared lock on its B+-tree parent; and a **range_search** can lock its first leaf node in shared mode if it already holds a shared lock on the leaf's B-tree parent, and can lock the subsequent leafs in shared mode along the leaf chain if it holds a shared lock on the predecessor in the chain. This technique is both serializable and deadlock-free.

   With knowledge on B-tree/B+-tree semantics, additional optimizations of index locking are not uncommon in practice, to name a few, *link technique* and *giveup tech-*

nique [41]. These variants improves index searching performance. However, they are beyond the scope of the TL protocol and the DAG protocol themselves.

2.2.10 Application of TL in concurrency control in Java

A recent research with TL involved is the implementation of *atomic block* in Java programming language [19]. Similar to a transaction, an atomic block executes an isolated sequence of method invocations on shared objects. Both 2PL and TL are tried for the concurrency control of atomic blocks.

As for TL, the lock trees in [19] are limited to binary trees with data items only on the leaf nodes. All the internal nodes are extra "virtual nodes". Some criteria for the "good" lock tree structures w.r.t. transaction concurrency are presented:

- The lower the root node of a transaction[7] is, the better the concurrency is.
- The acquisition of lock on a node must pay off, and concurrency can be optimal when transactions access all resources located below that node in the tree.
- Concurrency is increased if accesses to the data of a subtree are adjacent in a transaction.
- Concurrency is generally increased if the shared data items are accessed by multiple transactions in the same order.

Based on these criteria, [19] introduces a tree construction algorithm. The experimental results in [19] show that:

- When data contention is low, TL has exceptionally poor performance.
- When contention is high, TL performs empirically a little better than 2PL.
- When data are organized in hierarchical structures (*e.g.*, XML), TL performs far better than 2PL.
- The runtime overhead of TL is higher than that of 2PL.

[7]The root node of a transaction is the lowest common ancestor in the lock tree of all the objects accesses by it.

2.3 Non-flat transaction models

The classical transaction model [8], also known as page model [53], views a transaction as a linear sequence of *reads* and *writes*. A *history* is made up by a set of transactions whose operations can be interleaved, and a *schedule* is a prefix of a history. Two operations are *conflict* if they are issued by two different transactions upon the same data item and at least one of them is a write. The main standard of correctness is *serializability*.

Because database design is separated into multiple levels, and because of the introduction of abstract data types (ADTs) and object databases, it is natural and beneficial to extend this classical transaction model to richer non-flat models w.r.t. operations, conflict relations, and correctness. Some non-flat transaction models are reviewed as follows.

2.3.1 Nested transactions

Although many related models and algorithms are proposed earlier (*e.g.*, [38]), it is in [6] and [7] that a general formal model for *nested transactions* is given. The concepts of *computation*, *transaction/subtransaction*, and *computation forest* (*CF*) are defined, and the axioms each concept must satisfy are provided. In a tree of a CF, each transaction can call some (sub)transactions under it as its operations and can be called by another transaction above it. The papers extend serializability to this model. As defined, a CF is *serial* if there is a total order on its roots and on the children of each node, and a CF is *serializable* if it is equivalent to some serial CF. Moreover, the papers are more interested in the *order-preserving* serializability, in which a correct CF is not only equivalent to a serial CF but also has the same order of the root transactions as the serial one. So the correctness of a CF can be proved by showing it can be transformed to a serial one by applying finite steps of equivalence-preserving and order-preserving transformations, including (1) commutativity-based reversal of leafs and (2) reduction and expansion of subtransactions.

2.3.2 Multilevel transactions

[52] studies a restricted version of nested transaction model, *multilevel transaction*. It has the following features:

1. All paths from the root of any transaction tree to any of its leaf are of the same length. The nodes of the same distance from the roots form a *level*.

2. On each level L_i, the conflict relation CON_i between operations is defined.

3. Transactions are *conflict-faithful* [53], *i.e.*, if two operations are conflict at a certain level, at any lower level there must be a pair of conflict operations that are their descendants respectively.

4. The model initially specifies $<$ as a partial order only at the leaf level[8] in a schedule. For two conflict leaf operations, they must be ordered by $<$. And two non-leaf operations o_1 and o_2 are $<$-ordered, say $o_1 < o_2$, if and only if for each leaf descendant $o_1{}'$ of o_1 and each leaf descendant $o_2{}'$ of o_2, $o_1{}' < o_2{}'$ holds. Derivation of the order of internal operations from the leaf order is called *tree-consistent node ordering* [53].

The paper defines *quasi-order* $\leq_i$ at each level L_i based on the relationships $\leq_{i-1}$ and CON_{i-1} at the level L_{i-1} below it. For two L_i-operations o_1 and o_2, $o_1 \leq o_2$ if and only if there are a child $o_1{}'$ of o_1 and a child $o_2{}'$ of o_2 at L_{i-1} such that $(o_1{}', o_2{}') \in CON_{i-1}$ and $o_1{}' \leq_{i-1} o_2{}'$. And at the leaf level, $\leq_0 = <_0 \cap CON_0$. A multilevel schedule is *serializable* if and only if $\leq_i$ at each level L_i is acyclic. For a serializable schedule, $\leq_i$ specifies the order of L_i-operations in the serial schedule to which it is equivalent[9].

An interesting property of this model is that the multilevel serializability can be related to the classical flat transaction serializability between adjacent levels. An n-level schedule is serializable if all level-by-level schedules of adjacent levels are serializable, although

[8] In the nested transaction model in [6] and [7], the partial order $<$ is general among all operations in the CF.

[9] In the model, it can be concluded that $<_i \cap CON_i \subseteq \leq_i$. The serializability only requires $\leq_i \cup (<_i \cap CON_i)$ at each level L_i to be acyclic, but does not exclude the cycles in $\leq_i \cup <_i$. So the serializability is not order-preserving.

the inverse does not hold. This provides a practicable way to implement correct multilevel schedulers which allows independent concurrency control techniques (*e.g.*, locking protocols) at different levels.

2.3.3 Object transactions

The *object transaction model* is given in [53]. It models transactions with multiple abstraction levels with the lowest one being the page reads and writes. The object model inherits many important features from the multilevel transaction model in [52], but generalizes to arbitrary trees instead of trees with leafs of the same distance from the roots. The definitions below are from [53].

In the object transaction model, each transaction tree t_i is a pair $(O_i, <_i)$ with O_i as the operation node set and $<_i$ the partial order on its leaf level nodes. An *object model history* $(O_s, <_s)$ is a partial ordered forest of transaction trees, in which the node set $O_s = \bigcup O_i$ and the partial order $<_s$ satisfies $(\bigcup <_i) \subseteq <_s$. As in the multilevel transaction model [52], any pair of conflict leaf operations, which are issued by different transactions on the same data item and at least one of which is write, must be ordered by $<_s$. The $<_s$-order of internal nodes is derived from the $<_s$-order of all their respective leaf nodes, following the tree-consistent ordering (see Section 2.3.2). An *object model schedule* is a prefix of an object history. The multilevel history/schedule in [52], which requires leafs to have the same depth, is renamed as *layered history/schedule* in [53].

An object model schedule is *serial* if

- its roots are totally ordered, and
- for each root and for each $i > 0$, the descendants with distance i from the root are totally ordered.

A related concept is *isolated subtree*. A node p and the corresponding subtree in an object model schedule s are *isolated* if,

1. for all nodes q other than ancestors or descendants of p, for any leaf w of q, either $p <_s w$ or $w <_s p$, and,

2. for each $i > 0$ the descendants of p with distance i from p are totally ordered.

The correctness of an object model schedule is defined by its equivalence to a serial object model schedule. Similar to the commutativity-based reversal and reduction/expansion in [6] and [7], [53] defines a set of equivalence-preserving transformation rules upon a schedule. These rules are:

1. *Commutative rule:* The order of two ordered leaf operations p and q with order $p <_s q$ can be reversed provided that:
   (a) both the subtrees rooted at p and q are isolated and adjacent (no other operations r such that $p <_s r <_s q$);
   (b) the operations belong to different transactions, or if they belong to the same transaction t_i, the reversal does not contradict the specified order $<_i$ of t_i.
   (c) the operations p and q do not have ancestors p' and q' respectively which are noncommutative and total ordered in $p' <_s q'$.
2. *Ordering rule:* Two unordered leaf operations p and q can arbitrarily be ordered if p and q are commutative.
3. *Tree pruning rule:* An isolated subtree can be pruned in that it is replaced by its root.

An object history is said to *tree-reducible* if it can be transformed to a total order of roots by applying the above rules finitely many times. An object model schedule is *tree-reducible* if its committed projection is tree-reducible. Tree reducibility specifies the correctness of an object model schedule [53].

2.3.4 Saga model

The *saga model* [20] is originally proposed for *long lived transactions* (*LLTs*), for which strict atomicity is both costly and unnecessary. A *saga* is an LLT that is realized by a sequence of subtransactions, which can be interleaved with other transactions. After each subtransaction finishes, its resources are released without waiting for the completion of the

saga. However, a saga still needs to be treated as an execution unit. If a subtransaction is aborted and the saga can not complete successfully, the partial result of the already committed subtransactions of the saga is unacceptable. The cleanup requires a compensation transaction for each subtransaction. Suppose a saga has subtransactions $T_1, \cdots, T_i, \cdots, T_n$ and each T_i has a compensation C_i. A successful saga is exactly the sequence

$$T_1, \cdots, T_n$$

with each T_i committed. If $T_1, \cdots, T_{i-1}$ have already committed but something goes wrong with T_i, the sequence becomes

$$T_1, \cdots, T_{i-1}, T_i(aborted), C_{i-1}, \cdots, C_1$$

in which the compensations are executed in reverse order to undo those subtransactions that have been finished. The saga model has only two levels (saga level and subtransaction level), and the partial result of a saga is visible to other transaction (even if the saga is aborted, it is visible before the compensations are done). It is a simplified and relaxed non-flat transaction model suitable for LLTs that consist of a sequence of comparably independent steps.

Our embedded control program case study of Linux kernel source code shows the implementation of the system call **fork** also follows the saga model (see Appendix A.1).

Chapter 3

Applying TL to compiled database applications

3.1 Definitions

For compiled data applications, each transaction type is modelled as a finite state machine, in which each state represents an operation on a data item, and each directed arc indicates a transition from an operation to a next one. A state may have more than one direct successor, each with a probability assigned to the arc that leads to it. We assume only write operations at each state. The formal definition of a transaction type is:

Definition 3.1.1. *Let D be a* data set *that consists of data items in the database, and each data item is identified with a name.*

A transaction type *T is defined as $T = \langle N, s, F, A, data, cost, prob \rangle$.*

- *N is a set of states, each representing an operation.*
- *$s \in N$ is a starting state.*
- *$F \subseteq N$ is a set of terminating states.*
- *$A \subseteq (N - F) \times N$. A is a set of transition arcs from states to states. Terminating states have no outgoing arcs.*

- *data is a function from N to D. For each $n \in N$, $data(n)$ is the data item accessed at state n.*
- *cost is a function from N to nonnegative real number set. $cost(n)$ is the accessing cost (time duration, number of instructions, etc.) of $data(n)$ by state n.*
- *prob is a function from A to real numbers between 0 and 1. For each arc $\langle n_1, n_2 \rangle \in A$, $prob(\langle n_1, n_2 \rangle)$ is the probability that a transaction (See Definition 3.1.2) goes from n_1 to n_2. For each fixed $n \in (N - F)$,*

$$\sum_{\langle n,n' \rangle \in A} prob(\langle n, n' \rangle) = 1.$$

Based on the notations of the transaction type, we can define an individual transaction as follows:

Definition 3.1.2. *A* transaction *t of transaction type $T = \langle N, s, F, A, data, cost, prob \rangle$ is a sequence $n_0, \cdots, n_k$, where $n_i \in N$ for $0 \leq i \leq k$. Moreover, $n_0 = s$, $n_k \in F$, and for each adjacent pair n_{i-1}, n_i such that $1 \leq i \leq k$, $\langle n_{i-1}, n_i \rangle \in A$.*

Example 3.1.1. *An example of a transaction type is given in Figure 3.1. The starting state is n_1, and the terminating states are n_5, n_6, n_7, n_8. The data items and costs of the states are shown in the state boxes, separated by "/". The probabilities are attached to the arcs. An example of a transaction of the transaction type is $n_1, n_2, n_2, n_3, n_4, n_3, n_6$.*

In a compiled database application, there is a set of predefined transaction types, each with a probability. Such a set of transaction types is defined as a *transaction system.*

Definition 3.1.3. *A* transaction system *$S = \langle D, TS, prob \rangle$ is characterized by a set of transaction types $TS = \{T_1, \cdots, T_k\}$, on a data set D, and a function prob from TS to real numbers between 0 and 1. For each $T_i \in TS$, $prob(T_i)$ specifies the probability that a transaction of the transaction type T is chosen when it is to select the next transaction to execute. $\sum_{T_i \in TS} prob(T_i) = 1$. Moreover, $D = \bigcup_{T_i \in TS} \{data_i(n) \mid n \in N_i\}$, where each $T_i = \langle N_i, s_i, F_i, A_i, data_i, cost_i, prob_i \rangle$.*

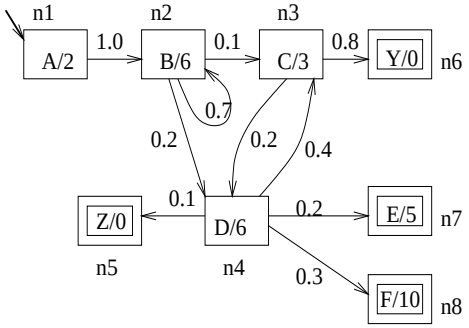


Figure 3.1: An example of a transaction type

3.2 Runtime locking and unlocking algorithms

3.2.1 Definitions of lock trees

We first define a *lock tree* as follows:

Definition 3.2.1. *A* lock tree $LT = \langle D, E \rangle$ *w.r.t. a transaction system* $S = \langle D, TS, prob \rangle$ *is a tree with a node set* D *and an edge set* E*. When we know* S *in the context, we call such a tree a* global lock tree*.*

Since there is a one-to-one mapping between the data item set and the tree node set, we define them as the same in Definition 3.2.1. In this paper, we will use the terms "data item" and "tree node" interchangeably. Moreover, it is unnecessary for a transaction to know the whole global lock tree. To manage tree locking, a transaction only needs to keep a part of the global lock tree that covers all data items its transaction type accesses.

Definition 3.2.2. *A* cover *of a transaction type* $T = \langle N, s, F, A, data, cost, prob \rangle$ *in a global lock tree* $GLT = \langle GD, GE \rangle$*, denoted as* $CLT = \langle CD, CE \rangle$*, is a tree such that* $CD \subseteq GD$*,* $CE \subseteq GE$*, and* $\{data(n) \mid n \in N\} \subseteq CD$*.*

In-transaction nodes *are the nodes* $\{d \in CD \mid \exists\ n \in N$ *such that* $data(n) = d\}$*. The remaining nodes in* CD *are* non-in-transaction nodes*. A cover of* T *in* GLT*, denoted as* $LLT = \langle LD, LE \rangle$*, is the* minimal cover *of* T *in* GLT *if there exists no cover of* T *in* GLT*, say,* $LT' = \langle D', E' \rangle$ *such that* $D' \subset LD$*,* $E' \subset LE$*. When the context is clear, we call such a minimal cover a* local lock tree*.*

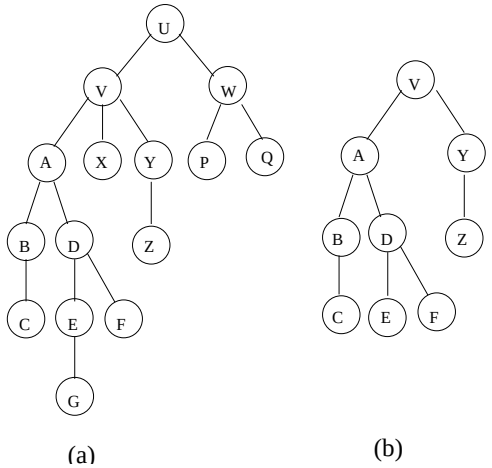


Figure 3.2: A global lock tree (a) and a local lock tree (b)

From Definition 3.2.2, the minimal cover of T in GLT must be unique. And all its leaf nodes must be in-transaction nodes.

Example 3.2.1. *For a global lock tree in Figure 3.2(a), the local lock tree w.r.t. the transaction type in Figure 3.1 is shown in Figure 3.2(b). In the local lock tree, all nodes but v are in-transaction nodes.*

For each transaction type, we store its local lock tree. The tree nodes with the same name in all local lock trees share a binary semaphore. Therefore, locking and unlocking a single node can be implemented with the **p** and **v** semaphore operations.

3.2.2 Locking and unlocking algorithms

We remind the reader that the TL operates as follows [27]:

(TL1) A transaction must lock a data item before it operates on it.

(TL2) Except for the first data item that a transaction locks, a data item can be locked by a transaction only if the transaction is currently holding a lock on its parent in the tree.

(TL3) Once a transaction releases a lock on a data item, it can never lock it again.

Under TL rules, there is still flexibility to choose the exact time of locking and unlocking. When the transaction types are fixed, the ideal policy that favors concurrency is the *late-locking/early-unlocking* policy:

1. Locking a tree node as late as possible.

2. When condition 1 is satisfied, unlocking a tree node as early as possible.

When a transaction wants to access a data item at a state, if it has not locked the corresponding tree node yet, it must lock it before accessing the data. It also retrieves all ancestors of the node in the local lock tree that it does not currently lock, and locks them in top-down order before it locks the node itself. Therefore, a transaction locks a tree node only when the transaction needs to access its data item or needs to lock it in order to further lock one of its children. The algorithm for locking is Algorithm 3.2.1, which satisfies the late-locking requirement.

Algorithm 3.2.1. *Locking the data item d accessed at a state:*

Require: *a set Locked that contains the tree nodes currently locked by the transaction.*

```
initialize an empty stack;
while d ∉ Locked do
    push d to the stack;
    if d is root of the local tree then
        break;
    else
        d ← d.parent;
    end if
end while
while the stack is not empty do
    pop d from the stack;
    Locked ← Locked ∪ {d};
    lock node d;
end while
```

Unlocking is more complicated than locking. We specify the unlocking conditions. From the rules TL2 and TL3, a transaction T can unlock a node d only if,

(U1) T will not access d in the future, and

(U2) for each child c of d, either c has already been locked, or all nodes in the subtree rooted at c will never be accessed by T in the future.

Condition U1 is formalized by the following definition.

Definition 3.2.3. *Given a transaction type $T = \langle N, s, F, A, data, cost, prob\rangle$ along with its local lock tree $LLT = \langle LD, LE\rangle$, for a state $n \in N$ and a data item $d \in LD$, if on all paths from n to all terminating states in F, each state n' satisfies $data(n') \neq d$, we say that d is* unreachable *from state n. The data set that are unreachable from state n is denoted as $UR(n)$ (*unreachable set*).*

The UR sets are inherent to the transaction type and can be computed at compile time. Calculating $UR(n)$ for each n is trivial. Therefore, at run time, testing both conditions U1 and U2 amounts to testing membership of the UR sets. If T unlocks d at the earliest state that satisfies both U1 and U2, the early-unlock requirement is met. However, we realize some difficulties in applying U1 and U2 directly:

- The size of $UR(n)$ is usually big, and it takes much memory to store $UR(n)$. Moreover, we only care about unlocking the data items that are being locked by the transaction.
- Checking U2 may require going over the whole subtree, which can be time consuming.

To reduce the storage overhead of $UR(n)$, we define two sets $UL(n)$ and TUL as follows.

Definition 3.2.4. *For a state $n \in N$ in a transaction type $T = \langle N, s, F, A, data, cost, prob\rangle$, its* transaction-type-unlockable set*, denoted as $UL(n)$, is a set of data items. Each $d \in UL(n)$ satisfies:*

1. *There exists some state n' that lies on a path from s to n, such that $d = data(n')$.*
2. *$d \in UR(n)$.*

3. *On each path from each n' (specified in 1) to n, there is no state n'' between n' and n such that $d \in UR(n'')$.*

Suppose a transaction of the transaction type T has gone through states n_1=s, n_2,$\cdots$, n_{k-1}, and is currently at state n_k, the total transaction-unlockable set *for the transaction, denoted as TUL, is defined as $TUL = \bigcup_{i=1,\cdots,k} UL(n_i)$.*

Intuitively, the elements in $UL(n)$ are those data items possibly locked by the transaction prior to the state n, and n is the *earliest* state from which these data items will never be accessed by the transaction. $UL(n)$ can also be computed at compile time. TUL is a runtime set corresponding to an executing transaction and can be maintained incrementally. When the transaction reaches a new state n, $TUL \leftarrow TUL \cup UL(n)$. From Definition 3.2.4, $UL(n) \subseteq TUL \subseteq UR(n)$ holds at each state n. Usually the inclusion is strict, and $UL(n)$ is much smaller than $UR(n)$ (See Example 3.2.2). We can store $UL(n)$ for each state n, and use TUL to approximate $UR(n)$ at runtime. This saves considerable storage space.

Example 3.2.2. *For the transaction type in Figure 3.1 and the local lock tree in Figure 3.2(b), the unreachable sets UR and transaction-type unlockable sets UL for all states are shown in Table 3.1.*

If a transaction follows the sequence n_1, n_2, n_3, n_4, n_3, n_4, n_7, the incrementally computed total transaction-unlockable set TUL at each state is shown in Table 3.2. TUL is often smaller than $UR(n)$. For example, at state n_7, TUL does not include F, Y, and Z that are never accessed by the transaction, while they are in $UR(n_7)$. Moreover, TUL never contains V, which corresponds to a non-in-transaction node in the tree, while all UR sets include it.

Next, we use *weak tree unlockability* to save the runtime cost of testing the unlocking condition U2.

Definition 3.2.5. *At a state of a transaction, a data item d is* weakly tree-unlockable *if each child c of d in the local lock tree either*

1. *has been locked by the transaction before, or*

state	UR	UL
n_1	$\{V\}$	$\{\}$
n_2	$\{V, A\}$	$\{A\}$
n_3	$\{V, A, B\}$	$\{B\}$
n_4	$\{V, A, B\}$	$\{B\}$
n_5	$\{V, A, B, C, D, E, F, Y\}$	$\{C, D\}$
n_6	$\{V, A, B, C, D, E, F, Z\}$	$\{C, D\}$
n_7	$\{V, A, B, C, D, F, Y, Z\}$	$\{C, D\}$
n_8	$\{V, A, B, C, D, E, Y, Z\}$	$\{C, D\}$

Table 3.1: Unreachable and transaction-type-unlockable sets at each state of the transaction type in Figure 3.1

state	*TUL up to the state*
n_1	$\{\}$
n_2	$\{A\}$
n_3	$\{A, B\}$
n_4	$\{A, B\}$
n_3	$\{A, B\}$
n_4	$\{A, B\}$
n_7	$\{A, B, C, D\}$

Table 3.2: Total transaction-unlockable set at each state of a transaction

2. is a leaf node and in TUL.

If a node is weakly tree-unlockable, it must satisfy U2. But the inverse does not hold. Testing weak tree unlockability is much cheaper because it stops at the level of direct children without testing the whole subtree underneath.

In terms of transaction-type unlockability and weak tree unlockability, the conditions under which a transaction T can unlock d are:

(U'1) $d \in TUL$ or d is a non-in-transaction node, and

(U'2) d is weakly tree-unlockable,

From the definitions, it is straightforward that U'1 implies U1, and U'2 implies U2. As a tradeoff to save storage and runtime overhead, U'1 and U'2 no longer guarantee early-unlocking but approximate it.

Based on U'1 and U'2, we design Algorithm 3.2.2 for unlocking. To improve the efficiency of weak tree unlockability testing, we attach a field **num_children_qualified** to each tree node. The field shows the number of children that satisfy the two conditions of weak tree unlockability. Each time a node is locked[1], or its data item is added to TUL at some state (if it is a leaf node), this field of its parent node is incremented by 1. If the value of **num_children_qualified** equals to the number of children of the node, the node is weakly tree-unlockable. Therefore, we only need to do a number comparison instead of iterating over all children.

Algorithm 3.2.2. *Unlocking data items, called at each state n:*

Require: *a set Locked that contains the nodes currently locked by the transaction.*

```
1: {Update the TUL:}
2: for each e ∈ UL(n) do
3:   if e ∉ TUL then
4:     TUL ← TUL ∪ {e};
5:     if (e ∉ LockSet) and (e is a leaf) then
6:       e.parent.num_children_qualified++;
```

[1]This can be done by adding a statement that increments the **num_children_qualified** value of the parent node by 1 after line 13 in Algorithm 3.2.1.

```
7:      end if
8:    end if
9:  end for
10: for all d ∈ Locked do
11:   {Testing the unlocking conditions:}
12:   if d is an in-transaction node then
13:     if d ∈ TUL then
14:       if d.num_children_qualified == d.num_children then
15:         Locked ← Locked \ {d};
16:         unlock node d;
17:       end if
18:     end if
19:   else
20:     if d.num_children_qualified == d.num_children then
21:       Locked ← Locked \ {d};
22:       unlock node d;
23:     end if
24:   end if
25: end for
```

Algorithm 3.2.2 is invoked when a transaction reaches a state, immediately before Algorithm 3.2.1 is called to lock the data item accessed. We further modify Algorithm 3.2.1 such that once we lock a node along the top-down tree path, we try unlocking its parent by testing the conditions U'1 and U'2 on it. When the transaction finishes at a terminating state, all remaining locks are released.

Example 3.2.3. *Following Example 3.2.2, under our locking and unlocking algorithms, the steps at each state of the transaction* n_1, n_2, n_3, n_4, n_3, n_4, n_7 *are presented in Table 3.3. We use* **l** *to represent a locking step,* **u** *an unlocking step, and* **a** *a data accessing. Please note the* TUL *sets at the states in the transaction are shown in Table 3.2.*

state	*steps*
n_1	$\mathbf{l}(V), \mathbf{l}(A), \mathbf{a}(A)$
n_2	$\mathbf{l}(B), \mathbf{a}(B)$
n_3	$\mathbf{l}(C), \mathbf{u}(B), \mathbf{a}(C)$
n_4	$\mathbf{l}(D), \mathbf{u}(A), \mathbf{a}(D)$
n_3	$\mathbf{a}(C)$
n_4	$\mathbf{a}(D)$
n_7	$\mathbf{u}(D), \mathbf{l}(E), \mathbf{a}(E), \mathbf{u}(V), \mathbf{u}(D), \mathbf{u}(E)$

Table 3.3: Steps at each state of a transaction

3.2.3 Correctness of the runtime algorithms

We formally prove that our runtime algorithms obeys the TL protocol. The satisfaction of the TL1 rule is trivial. We show that Algorithm 3.2.1 and 3.2.2 also satisfy TL2 and TL3.

Lemma 3.2.1. *Locking with Algorithm 3.2.1 obeys the rule TL2.*

Proof. By Algorithm 3.2.1, a transaction T always at first locks the root of the local lock tree. From the definition of the local lock tree (Definition 3.2.2), all data items that T may access are reachable from the root of the local lock tree.

If T is going to lock a data item d when d's parent d' has not been locked by T, by Algorithm 3.2.1, T will lock d' and then holds the lock on d' when it locks d. So TL2 is satisfied.

Even if we modify Algorithm 3.2.1 in that it tries to unlock the parent after it locks the child by testing unlocking conditions, it does not unlock the parent before it locks the child. So it does not change the satisfaction of TL2. □

Lemma 3.2.2. *Following Algorithm 3.2.1 and the unlocking condition U1, if a transaction T has to lock data item d immediately before accessing it, it is the first time T locks d.*

Proof. By contradiction. Assume T has locked d, then unlocked it at state n', and is now requesting the lock on d again at a later state n. From condition U1, T can unlock d only if T will never access d after state n', including at state n. So we reach the contradiction. □

Lemma 3.2.3. *Locking with Algorithm 3.2.1 and unlocking under conditions U1 and U2 obey the rule TL3.*

Proof. By contradiction. Assume that once a transaction t unlocks data item d, it is possible that t locks it again. From condition U1, t will not access d again. So it will not lock d again for accessing d. From Algorithm 3.2.1, the only possibility is that t locks d again to lock and access a descendant of d in the local lock tree, say, d''. From Lemma 3.2.2, t locks d'' for the first time. Without loss of generality, we suppose the path from d to d'' consists of $d_0 = d$, $d1$, $\cdots$, $d_k = d''$. From Algorithm 3.2.1, t is current holding lock on none of $d_0, \cdots, d_k$. As t has unlocked d_0, from condition 2, it must have locked d_1 before it unlocks d_0. Therefore, t must have locked and then unlocked d_1, and now is locking d_1 again. By induction, t must have locked $d_k = d''$ before. So we have a contradiction. □

Theorem 3.2.1. *Locking with Algorithm 3.2.1 and unlocking under conditions U1 and U2 guarantee both serializability and deadlock-freedom.*

Proof. From Lemma 3.2.1 and Lemma 3.2.3, and the features of TL in [27]. □

Corollary 3.2.1. *Locking with Algorithm 3.2.1 and unlocking with Algorithm 3.2.2 guarantee both serializability and deadlock-freedom.*

Proof. Algorithm 3.2.2 applies unlocking conditions U'1 and U'2. U'1 implies U1, and U'2 implies U2. □

3.3 Compile-time processing

In Section 3.2, for each transaction type, runtime locking/unlocking relies on two types of data structures: the transaction-type-unlockable sets and the local lock tree. For a predefined transaction system, these structures can be generated at compile time.

3.3.1 Generating unlockable data sets

First, we derive the algorithm to compute the transaction-type-unlockable sets UL. In a transaction type $T = \langle N, s, F, A, data, cost, prob \rangle$, only N, A, s, F are relevant to this

processing. We define $G(T) = \langle N, A \rangle$ as a directed graph whose nodes are the states in T and whose arcs are transitions in T. $G(T)$ has a source node s and a set of sink nodes F.

For two nodes $n_1, n_2 \in N$, we say that n_2 is *reachable* from n_1 if n_2 and n_1 are the same node, or there is a directed path from n_1 to n_2 in $G(T)$; and n_2 is *after* n_1 if n_2 is reachable from n_1 but not vice versa.

For each $n \in N$, we define a set $RL(n) = \{n_k \in N \mid n_k \text{ is after } n \text{ and } data(n) \in UL(n_k)\}$. For a data item d, we can conclude that

$$\bigcup_{\substack{n' \in N, \\ data(n')=d}} RL(n') = \{n \in N | d \in UL(n)\}. \tag{3.1}$$

So once we have the RL sets for all states, we can build their UL sets. Next, we show how to build the RL sets.

We denote $L(n)$ as the set of all nodes n_i such that:

1. n_i is reachable from n, and,

2. there is an n_k reachable from n_i such that $data(n_k) = data(n)$.

Consequently, if $n_i \in L(n)$, then $n_i \notin RL(n)$. Moreover, immediately from the definitions of $UL(n)$, $RL(n)$, and $L(n)$, the following proposition holds.

Proposition 3.3.1. *For each $n_j \notin L(n)$, $n_j \in RL(n)$ if and only if there exists an $n_k \in L(n)$ and $\langle n_k, n_j \rangle \in A$.*

Based on Proposition 3.3.1, we can derive Algorithm 3.3.1 that computes the RL and UL sets for all states. In each iteration of the main loop (line 5), the algorithm processes a state n by calling the **dfs_mark** procedure. **dfs_mark** is based on the depth-first searching algorithm, and it computes a subset of $L(n)$, defined as $L'(n) = \{n_i \in L(n) \mid n_i \text{ is reachable from } n \text{ without passing any } n_k \neq n \text{ such that } data(n_k) = data(n)\}$. From Proposition 3.3.1, for a node $n_i \in L'(n)$, its direct successor n_k can only be one of the three following cases: (1) $n_k \in L'(n)$, (2) $data(n_k) = data(n)$, or (3) $n_k \in RL(n)$.

After **dfs_mark** returns, all nodes in $L'(n)$ are marked. Then the main algorithm continues to build a node set $RL'(n) = \{n_i \notin L'(n) \mid data(n_i) \neq data(n)$ and there is an

$n_k \in L'(n)$ such that $(n_k, n_i) \in A\}$ in the rest of the iteration. We can see

$$\begin{aligned}(RL(n) - \bigcup_{\substack{n_i \text{ is after } n,\\ data(n_i)=data(n)}} RL(n_i)) \\ \subseteq RL'(n) \subseteq RL(n).\end{aligned} \tag{3.2}$$

For each $n_i \in RL'(n)$, the main program puts $data(n)$ into $UL(n_i)$. Based on (3.1) and (3.2), only correct data items are added to $UL(n_i)$, so the algorithm is sound.

Furthermore, we show the completeness of the algorithm. Because the algorithm loops over all state nodes, the elements in $RL(n)$ that are missing in $RL'(n)$ will be compensated when other nodes that access the same item are processed in the later iterations. Therefore, for each data item d,

$$\bigcup_{\substack{n\in N,\\ data(n)=d}} RL(n) = \bigcup_{\substack{n\in N,\\ data(n)=d}} RL'(n) \tag{3.3}$$

holds. By the end of the algorithm, d is added into the UL sets at all nodes in $\bigcup_{\substack{n\in N,\\ d=data(n)}} RL(n)$.

Algorithm 3.3.1. *Compute $UL(n)$ for each state n:*

compute the strongly connected components of $G(T)$ [16]
for** each state $n \in N$ **do
  $UL(n) \leftarrow \{\}$;
end for
for** each state $n \in N$ **do
  for** each state $n' \in N$ **do
    $n'.marked \leftarrow false$;
    $n'.visited \leftarrow false$;
  end for
  $n.marked \leftarrow true$;
  call function $dfs\_mark(data(n), n)$;
  for** each marked n' **do
    for** each successor n'' of n' **do
      if** n'' is not marked and $data(n'') \neq data(n)$ **then
        $UL(n'') \leftarrow UL(n'') \cup \{data(n)\}$;

```
16:         end if
17:       end for
18:     end for
19: end for
```

Function dfs_mark does a depth-first searching, and marks nodes in $L'(n)$. The return value indicates whether n is in $L'(n)$.

```
1: function dfs_mark(d, n) returns boolean
2: n.visited ← true;
3: flag ← false;
4: if n ∈ F then
5:   return(false);
6: else
7:   for each successor n' of n do
8:     if n'.visited then
9:       if n'.marked then
10:        flag ← true;
11:      end if
12:    else
13:      if data(n') = d then
14:        flag ← true;
15:      else
16:        flag ← flag ∨ dfs_mark(d, n');
17:      end if
18:    end if
19:  end for
20: end if
21: if flag then
22:   n.marked ← true;
23:   if n is in a strongly connected component C then
24:     for each n'' ∈ C do
25:       if data(n'') ≠ d then
```

```
26:            n''.marked ← true;
27:          end if
28:        end for
29:      end if
30:    end if
31:  return(flag);
```

3.3.2 Generating lock trees

Building a global lock tree that "fits" the transaction system is important for the efficiency of the TL approach. We find some rough categories for tree fitness:

(C1) A lock tree should favor the transaction types with higher probabilities in a transaction system.

(C2) A lock tree should favor the transactions with higher probabilities under the same transaction type.

(C3) For those data items accessed at the states that are close in a transaction type, it is preferable that their corresponding tree nodes are also close, *e.g.*, as parent and child, or children under the same parent.

(C4) A transaction should lock as few non-in-transaction nodes as possible in the lock tree.

Our idea is at first building individual reference trees from each transaction type, then merging them into a global lock tree, and finally projecting the global lock tree to individual transaction types to obtain local lock trees. Algorithm 3.3.2, which is based on depth-first searching over a finite state machine, builds the reference tree for a transaction type T. It always expands the unvisited successor with the highest probability of the current state. This follows the category C2. The reference tree covers all data items accessed by the transaction type.

Algorithm 3.3.2. *Given a transaction type* $T = \langle N, s, F, A, data, cost, prob \rangle$*, build a reference tree* t*:*

```
1: add data(s) to t as root;
```

2: *call procedure build_tree(s, t);*

1: ***procedure*** *build_reference_tree(n, t);*
2: *$n.visited \leftarrow true$;*
3: ***while*** *there is unvisited successor of n in $G(T)$* ***do***
4: *choose the unvisited successor c with the highest prob$(\langle n, c \rangle)$;*
5: ***if*** *data(c) is not a node in t* ***then***
6: *add data(c) as child of data(n) in t;*
7: ***end if***
8: *call procedure build_reference_tree(c, t);*
9: ***end while***

Example 3.3.1. *Applying Algorithm 3.3.2 to the transaction type in Figure 3.1 returns the tree in Figure 3.3.*

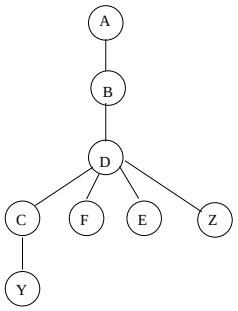


Figure 3.3: The tree built by Algorithm 3.3.2 from the transaction type in Figure 3.1

For a transaction system with a transaction type set $TS = \{trans_1, \cdots, trans_n\}$, after we build the reference trees from all $trans_i \in S$, we merge them into a global lock tree. We sort the trees in descending order of the probabilities of the corresponding transaction types, and get an array of trees $[tree_1, \cdots, tree_n]$. Then we build the global lock tree with Algorithm 3.3.3.

Algorithm 3.3.3. *Given an ordered array of trees $[tree_1, \cdots, tree_n]$, build a global lock tree GLT:*

```
1:  initialize GLT as a copy of tree_1;
2:  for i ← 2 to n do
3:    for each node d in tree_i do
4:      if d is not in GLT then
5:        if the parent of d in tree_i, d', is in GLT then
6:          add d to GLT as a child of d';
7:        else if d has a child d' in tree_i, and d' is the root of GLT then
8:          {d becomes the new root:}
9:          add d to GLT as the parent of d';
10:       else if d has a child d' in tree_i, and d' is a non-root node in GLT then
11:         find the parent of d' in GLT, d'';
12:         add d to GLT as a child of d'';
13:       else
14:         add d as the child of an arbitrary leaf node in GLT;
15:       end if
16:     end if
17:   end for
18: end for
```

Algorithm 3.3.3 initializes the global lock tree as the tree of the "most likely" transaction type (line 1), and processes other trees following decreasing order of the probabilities of the corresponding transaction types. By doing this, we try to favor the transaction types with higher probabilities (category C1) and avoid non-in-transaction nodes in their local lock trees (category C4). The rules for adding a new node into the global lock tree are reflected at line 5-14, which is to fulfill category C3 and to make the data items that are probably accessed at close states in a transaction type be also close in the global lock tree.

After the global lock tree is built, the algorithm to build the local lock tree for each transaction type is straightforward. First, the set of tree nodes $D = \{d'_1, \cdots, d'_n\}$ that correspond to the data items accessed by the transaction type belong to the local lock tree. Then the algorithm sets $r' \leftarrow d'_1$, and then iterates over $d'_2, \cdots, d'_n$. For each d'_i, it finds the lowest common ancestor r'' of r' and d'_i, and adds all nodes and edges between r'' and r' and between r'' and d'_i to the local lock tree. Then it resets $r' \leftarrow r''$ and continues to

process d'_{i+1}. When the loop is over, the algorithm finishes building the local lock tree.

3.3.3 Inserting lock steps into transaction types

Given the lock tree and the transaction-type-unlockable sets, we can generate the locking and unlocking programs following Algorithm 3.2.1 and 3.2.2 at compile time. However, the runtime CPU overhead of locking and unlocking is still high, as there are loops in the algorithms. Moreover, we still need to keep the local lock tree and unreachable sets for each transaction type, and the total transaction-unlockable set for each transaction, which incurs memory overhead. To save such costs, we further investigate how to generate all locking steps at compile time.

We extend the notion of the transaction type in Definition 3.1.1 with *lock steps*. A lock step is a step that locks or unlocks a data item on a transition arc[2]. Our task is inserting lock steps into the transaction types following the TL protocol. As it is compile-time processing, we can afford to use the bigger unreachable set UR and the slower but more accurate unlocking conditions (U1 and U2 in Section 3.2)[3].

Example 3.3.2. *Suppose we have a transaction type and its local in Figure 3.4(a). After we add lock steps into it, it becomes the transaction type in Figure 3.4(b), in which we use solid round nodes to represent lock states. The sequence of locking and unlocking of data items follows the TL protocol and late-lock/early-unlock policy.*

Adding lock steps incurs a new complication. For a certain non-start state n, there may be several paths from s to n. Each path may have accessed different data items, and thus result in different sets of data items locked by the transaction at n. The lock steps after

[2]To hold lock steps before the starting state and after the terminating states, we add a transition arc before the starting state, and one after each terminating state.

[3]Therefore, the locking conditions based on UL and TUL sets and weak tree unlockability in Algorithm 3.2.2 for runtime overhead saving are no longer necessary. It is also not necessary to calculate UL sets for the states (Section 3.3.1). Computing UR sets at all states at compile-time is trivial. Suppose the data set is D. To get $UR(n)$, we just initialize a set $VS \leftarrow \emptyset$, and do a searching (either breadth-first or depth-fist) from the state n. For each state n' visited in the searching, we add $data(n')$ to VS. After the searching is done, $UR(n) \leftarrow D \backslash VS$.

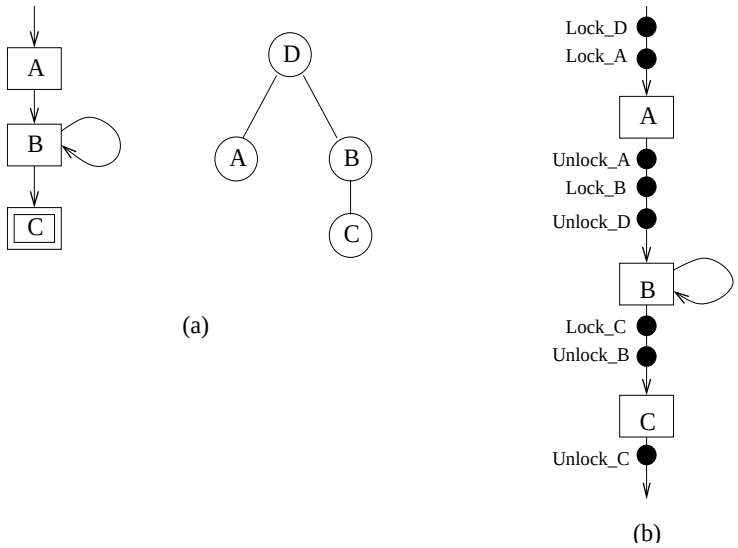


Figure 3.4: Adding lock states to a transaction type: (a) original transaction type and its local lock tree; (b) expanded transaction type.

the confluence state n, which depend on the locked data set at n, may be nondeterministic. The problem is illustrated in Example 3.3.3.

Example 3.3.3. *Suppose we have the transaction type and the local lock tree in Figure 3.5 (copied from Figure 3.1 and 3.2(b)). For a transaction goes through n_1, n_2, n_4, n_5, its sequence with lock steps added is Lock_V, Lock_A, n_1, Lock_B, n_2, Lock_D, Unlock_A, n_4, Unlock_B, Unlock_D, Lock_Y, Unlock_V, Lock_Z, Unlock_Y, n_5, Unlock_Z. And for another transaction that takes n_1, n_2, n_4, n_3, n_4, n_5, its sequence is Lock_V, Lock_A, n_1, Lock_B, n_2, Lock_D, Unlock_A, n_4, Lock_C, Unlock_B, n_3, n_4, Unlock_D, Unlock_C, Lock_Y, Unlock_V, Lock_Z, Unlock_Y, n_5, Unlock_Z. As the first transaction does not have the state n_3 and does not lock C, there is no lock step Unlock_C on the arc $\langle n_4, n_5 \rangle$. The second accesses C at n_3, so it has a lock step Lock_C on $\langle n_4, n_3 \rangle$, and an Unlock_C on $\langle n_4, n_5 \rangle$.*

To handle such problem, we need to expand a transaction type to make each possible path have a unique sequence of lock steps. A naive approach is shown in Figure 3.6. If from

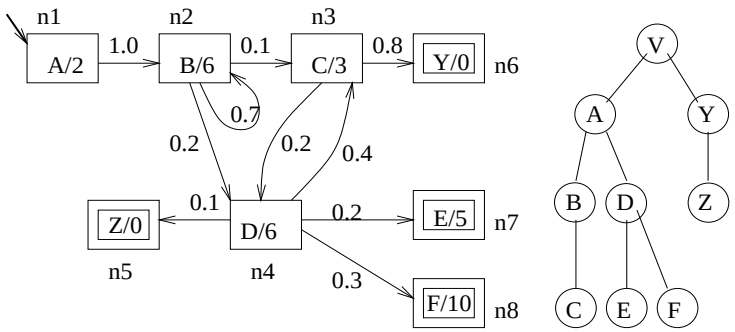


Figure 3.5: A transaction type and its local lock tree

s to n_m there are two paths, then from n_m on, we duplicate each state n_k to a new state n'_k and the path going through the duplicated states accordingly. Therefore, along each path in the expanded transaction type, the lock step sequence is deterministic. However, such expansion based solely on prefix paths may not be necessary. Intuitively, if the two paths from s to n_m in Figure 3.6(a) are of the same length, and access the same set of data items in the same order, the expansion is not necessary because the lock steps after n_m are the same. Next, we show that the number of different sets of locked items that a transaction may have at the confluent state n_m determines how many duplications we should have for n_m and the subsequent path.

Define the set of data items in the local lock tree being locked by a transaction t at a state n as $L_{t,n}$, and the set of data items in the local lock tree that have been locked (possibly already unlocked) by t at n as $V_{t,n}$. We also define the set of data items locked by t at a certain moment (possibly between two lock steps on the arc) as L_t, and the set of data items that have been locked (possibly already unlocked) by t at the moment V_t. On a transition arc $\langle n, n' \rangle$, a transaction t first tests on each data item $d \in L_{t,n}$ with following conditions:

(U1) $d \in UR(n')$;

(U2) for each child c of d in the local lock tree, either $c \in V_t$, or for all nodes c' in the subtree rooted at c, $c' \in UR(n')$.

t unlocks d if they are satisfied. The processing up to now is named as the *unlocking stage*.

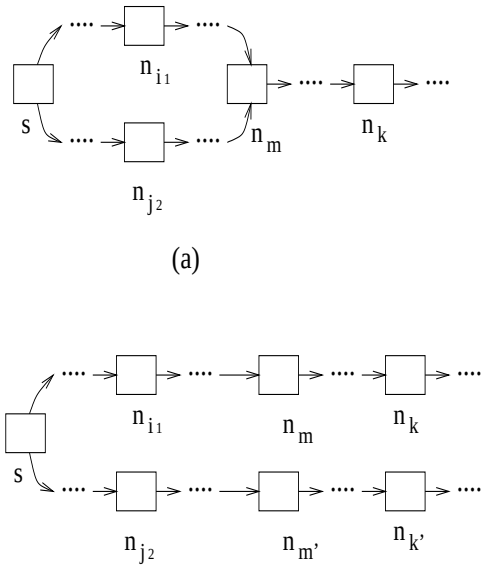


Figure 3.6: A naive expanding approach: (a) original transaction type, (b) expanded transaction type

If it is necessary, t enters the *locking stage*. It executes Algorithm 3.2.1 to lock $data(n')$. It is possible that t has to lock its ancestors in the top-down order. Furthermore, after t locks a tree node, it also tests the unlocking conditions on its parent and tries to unlock it.

For such processing, we prove Theorem 3.3.1.

Theorem 3.3.1. *Given the transaction type T in Figure 3.6(a), suppose transaction t_1 goes through $s, \cdots, n_{i_1}, \cdots, n_m, \cdots, n_k, \cdots$, and transaction t_2 goes through $s, \cdots, n_{j_2}, \cdots, n_m, \cdots, n_k, \cdots$, if $L_{t_1,n_m} = L_{t_2,n_m}$, then t_1 and t_2 have the same sequence of lock steps after n_m.*

Proof. First, we show that the sequence of lock steps of t_1 and t_2 are exactly the same between n_m and its successor n_{m+1}, and $L_{t_1,n_{m+1}} = L_{t_2,n_{m+1}}$. We restrict L_{t_1}, V_{t_1}, L_{t_2}, and V_{t_2} to the arc $\langle n_m, n_{m+1}\rangle$. Denote L_{t_1,n_m} and L_{t_2,n_m} as L_{n_m}.

Lemma 3.3.1. *For each d' on some path from the root to any $d'' \in L_{n_m}$, $d' \in V_{t_1,n_m} \cap V_{t_2,n_m}$.*

Proof. By Algorithm 3.2.1. □

Lemma 3.3.2. *During the unlocking stage, t_1 and t_2 unlock the same set of data items by testing the conditions U1 and U2 over L_{n_m}.*

Proof. During the unlocking stage, $V_{t_1} = V_{t_1,n_m}$, $V_{t_2} = V_{t_2,n_m}$, and V_{t_1} and V_{t_2} are not changed.

When t_1 and t_2 test the conditions on a data item $d \in L_{n_m}$, they have the same result on U1. However, it is possible that $V_{t_1,n_m} \neq V_{t_2,n_m}$. So they have different V_{t_1} and V_{t_2} for U2. To prove the lemma, it suffices to show such difference does not influence the testing result of U2.

Without loss of generality, we suppose there is a child c of d such that $c \in (V_{t_1,n_m} \backslash V_{t_2,n_m})$. As $c \notin L_{t_1,n_m}$, c has been unlocked by t_1 rightly prior to a state n_{i_1}. Either n_{i_1} is before n_{n_m}, or $n_{i_1} = n_{n_m}$. From the unlocking conditions, $c \in UR(n_{i_1})$. Since UR sets are always incremental along the path in the transaction type, $c \in UR(n_{m+1})$. Moreover, for each child c_1 of c, either (1) $c_1 \in V_{t_1,n_{i_1}}$, or (2) for all nodes c_2 in the subtree rooted at c_1, $c_2 \in UR(n_{i_1})$. We discuss the two cases below.

- For case (2), because UR sets are incremental, for all c_2 in the subtree rooted at c_1 (including c_1), all $c_2 \in UR(n_{m+1})$.
- For case (1), from Lemma 3.3.1, $c_1 \notin L_{n_m}$. So c_1 has also been unlocked by t_1 before n_m. Therefore, $c_1 \in UR(n_{m+1})$. Moreover, because $c \notin V_{t_2,n_m}$, $c_1 \in (V_{t_1,n_m} \setminus V_{t_2,n_m})$. By induction, for all data items c_3 in the subtree rooted at c_1, $c_3 \in UR_(n_{m+1})$.

Therefore, for each data item c' in the subtree rooted at c, $c' \in UR(n_{m+1})$. When testing the condition U2, if t_1 gets $c \in V_{t_1}$, then t_2 also finds that all node c' in the subtree rooted at c, $c' \in UR(n_{m+1})$. So such difference in V_{t_1,n_m} and V_{t_2,n_m} does not influence unlocking testing. □

Lemma 3.3.3. *During the locking stage, both t_1 and t_2 have the same sequence of lock steps.*

Proof. Because of Lemma 3.3.2, t_1 and t_2 have the same sequence of lock steps during the unlocking stage. At the end the unlocking stage, $L_{t_1} = L_{t_2}$. It is obvious that $data(n_{m+1}) \in L_{t_1}$ if and only if $data(n_{m+1}) \in L_{t_2}$. So either both t_1 and t_2 enter the locking stage, or neither does. When Algorithm 3.2.1 is called in the locking stage, for both t_1 and t_2, it puts the same sequence of data items $data(n_{m+1}) = d_l$, d_{l-1}, $\cdots$, d_1 in the local lock tree into the stack, and pops out the same data item d_1 to start locking.

After d_1 is locked, $L_{t_1} = L_{t_2}$ still holds, $V_{t_1} = V_{t_1,n_m} \cup \{d_1\}$, and $V_{t_2} = V_{t_2,n_m} \cup \{d_1\}$. Now the unlocking conditions are tested on the parent of d_1, d_0. In a way similar to the proof of Lemma 3.3.2, we can infer that t_1 unlocks d_0 if and only if t_2 unlocks d_0. After processing d_0, we still have $L_{t_1} = L_{t_2}$. The final steps on the arc are locking d_l and trying unlocking d_{l-1}. By induction, t_1 and t_2 have the same sequence of lock steps during the locking stage, and at the end of the locking stage $L_{t_1} = L_{t_2}$ remains true. □

By Lemma 3.3.2 and Lemma 3.3.3, t_1 and t_2 have the same sequence of lock steps on $\langle n_m, n_{m+1} \rangle$, and $L_{t_1,n_{m+1}} = L_{t_2,n_{m+1}}$. By induction, t_1 and t_2 have the same sequence of lock steps along the path after n_m. □

From Theorem 3.3.1, we can simulate transactions on the transaction type. We define $L_T(n) = \{L_{t_i,n} \mid t_i$ is a transaction of the transaction type T that passes $n\}$. Obviously,

$L_T(n)$ is a finite set. We duplicate the original state n to $|L_T(n)|$ new states, and duplicate all states and paths after n accordingly. Hence, in the expanded transaction type, each state n has $|L_T(n) = 1$. Now we give the formal definition of expansion.

Definition 3.3.1. *Given a transaction type $T = \langle N,\ s,\ F,\ A,\ data,\ duration,\ prob\rangle$, the* expansion *process to generate the new transaction type $T' = \langle N',\ s',\ F',\ A',\ data',\ duration',\ prob'\rangle$ must satisfies:*

1. *There is a duplication function dup_state from N to the powerset of N'. For an $n \in N$, dup_state(n) produces the set of duplicated states in N'. For any two distinct states $n_1,\ n_2 \in N$, $(dup\_state(n_1) \cap dup\_state(n_2)) = \emptyset$. $N' = \bigcup_{n\in N} dup\_state(n)$.*

2. *There is a duplication function dup_transition from A to the powerset of A'. For an arc $\langle n_1, n_2\rangle \in A$, dup_transition$(\langle n_1, n_2\rangle)$ produces the set of duplicated arcs in A'. And for each $\langle {n_1}', {n_2}'\rangle \in dup\_transition(\langle n_1, n_2\rangle)$, it must hold that ${n_1}' \in dup\_state(n_1)$ and ${n_2}' \in dup\_state(n_2)$. For two distinct transitions $a_1,\ a_2 \in A$, $dup\_transition(a_1) \cap dup\_transition(a_2) = \emptyset$. $A' = \bigcup_{a\in A} dup\_transition(a)$.*

3. *$F' = \bigcup_{n\in F} dup\_state(n)$.*

4. *If a state $n \in N$ has $|L_T(n)| = k$, $|dup\_state(n)| = k$ must hold in T'. Each $n' \in N'$ can only have $|L_{T'}(n') = 1$. For each $n' \in N'$, suppose L' is the unique element in $L_{T'}(n')$, there must be an $n \in N$,$n' \in dup\_state(n)$, and $L \in L_T(n)$. dup_state(s) has only one element, s'.*

5. *If for $\langle n_1,\ n_2\rangle \in A$, and $\langle {n_1}',\ {n_2}'\rangle \in A'$, ${n_1}' \in dup\_state(n_1)$, and ${n_2}' \in dup\_state(n_2)$, then $prob'(\langle {n_1}', {n_2}'\rangle) = prob(\langle n_1, n_2\rangle)$.*

6. *For each $n' \in dup\_state(n)$, $cost'(n') = cost(n)$.*

The 4th condition is derived from Theorem 3.3.1.

Example 3.3.4. *For the transaction type T in Example 3.3.3, the L_T values at the states are:*

$L_T(n_1) = \{\{V, A\}\}$,
$L_T(n_2) = \{\{V, A, B\}\}$,
$L_T(n_3) = \{\{V, A, C\}, \{V, C, D\}\}$,
$L_T(n_4) = \{\{V, B, D\}, \{V, C, D\}\}$,
$L_T(n_5) = \{\{Z\}\}$,
$L_T(n_6) = \{\{Y\}\}$,
$L_T(n_7) = \{\{E\}\}$,
$L_T(n_8) = \{\{F\}\}$.

We expand T to the transaction type T' in Figure 3.7. In the expanded transaction type, the duplicate_state mappings are:

$duplicate\_state(n_1) = \{n_1\}$,
$duplicate\_state(n_2) = \{n_2\}$,
$duplicate\_state(n_3) = \{n_4, n_5\}$,
$duplicate\_state(n_4) = \{n_3, n_9\}$,
$duplicate\_state(n_5) = \{n_6\}$,
$duplicate\_state(n_6) = \{n_{10}\}$,
$duplicate\_state(n_7) = \{n_7\}$,
$duplicate\_state(n_8) = \{n_8\}$.

Except the obvious one-to-one mappings, the $L_{T'}$ values are:

$L_{T'}(n_4) = \{\{V, A, C\}\}$,
$L_{T'}(n_5) = \{\{V, C, D\}\}$,
$L_{T'}(n_3) = \{\{V, B, D\}\}$,
$L_{T'}(n_9) = \{\{V, C, D\}\}$.

Based on Theorem 3.3.1 and Definition 3.3.1, Algorithm 3.3.4 expands a transaction type T to T'. It is similar to a breadth-first search. Starting from s, the algorithm duplicates an s' into N', calculates $L_{T'}(s')$, and put the triple $\langle s', s, L_{T'}(s')\rangle$ into a queue. When a triple $\langle n', n, L\rangle$ is dequeued, it means $n' \in duplicate\_state(n)$ and $L = L\_T'(n')$. For each $m \in N$ such that $\langle n, m\rangle \in A$, the algorithm computes a lock set $LS \in L_T(m)$ from $L_{T'}(n')$ (which has only one member). Then it checks each $m'' \in N'$ such that

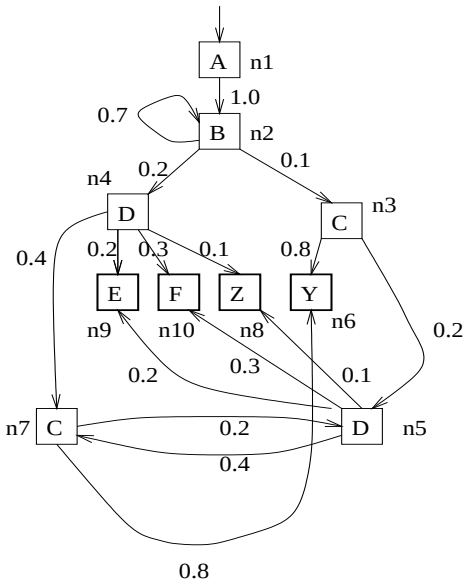


Figure 3.7: Expanded transaction type

$m'' \in duplicate\_state(m)$ to find one such that $L_{T'}(m'') = LS$. If there is such a m'', there is no need to duplicate m, and just a new arc $\langle n', m'' \rangle$ is added to A'. Otherwise, it duplicates m to a state m', adds m' into N', $\langle n', m' \rangle$ into A', and m' into $duplicate\_state(m)$, and enqueues the triple $\langle m', m, \{LS\} \rangle$ for further expansion.

In the algorithm, calculating $LS \in L_T(m)$ from $L_(T')(n')$ is in essence simulating a transaction t along the arc $\langle n, m \rangle$ following the locking/unlocking algorithms, Algorithm 3.2.1 and Algorithm 3.2.2 (with unlocking conditions modified). In the context of t, $L_(T')(n') = \{L_{t,n}\}$, and $LS = L_{t,m}$. The simulation adds lock steps while maintaining L_t as the result of the lock steps, and finally obtains $L_{t,m}$. Recall the unlocking conditions applied are:

(U1) $d \in UR(m)$;

(U2) for each child c of d in the local lock tree, either $c \in V_t$, or for all nodes c' in the subtree rooted at c, $c' \in UR(m)$.

As Algorithm 3.3.4 cannot maintain the $V_{t,n}$ and V_t sets, U2 is modified to an equivalent

condition in term of L_t and then applied in Algorithm 3.3.4:

(U2) for each c' in the subtree rooted at d, if $c' \notin UR(m)$, then either $c' \in L_t$, or there is a $d' \in L_t$ standing between d and c' on the path from d to c'.

Among the procedures called by Algorithm 3.3.4, **check_unlock** checks unlocking conditions, **remove_lock_set** tries to unlock a data item by calling **check_unlock**, **insert_lock_set** simulates the locking stage, and **build_L** simulates the whole processing on the transition arc.

Algorithm 3.3.4. *Given a transaction type $T = \langle N,\ s,\ F,\ A,\ data,\ duration,\ prob\rangle$, and a local lock tree LLT, expand to $T' = \langle N',\ s',\ F',\ A',\ data',\ duration',\ prob'\rangle$. For each state $n' \in N'$, there are two fields **lock_set** and **original_state**. The former is the member set in $L_{T'}(n')$, and the latter is the state $n \in N$ such that $n' \in dup\_state(n)$. Each arc $l\angle n\_1',\ n\_2'\rangle$ is attached with a sequence of lock steps, **lock_steps**. For each final state $n'_f \in F'$, there is a sequence **post_final_steps**, which releases the remaining locks after n'_f. T' also has a **pre_start_steps**, which does locking before accessing s'.*

Require: *an empty queue q;*

1: *generate a new node s';*
2: *put s' into T' as the starting state;*
3: *$N' \leftarrow \{s'\}$;*
4: *$s'.original\_state \leftarrow s$;*
5: *$cost'(s') \leftarrow cost(s)$;*
6: *$pre\_start\_steps \leftarrow$ empty sequence;*
7: *$s'.lock\_set \leftarrow \emptyset$;*
8: *{build $L_{T'}(s')$ and the lock steps before s': }*
9: *call procedure $build\_L(s,\ s'.lock\_set,\ pre\_start\_steps)$;*
10: *$enqueue(s', q)$;*
11: ***while*** *the queue q is not empty* ***do***
12: *$n' \leftarrow dequeue(q)$*
13: ***if*** *$n' \in F'$* ***then***
14: *{a final state:}*

```
15:     for all d ∈ n'.lock_set do
16:       add "unlock d" into n'.post_final_steps;
17:     end for
18:   else
19:     {not a final state: }
20:     n ← n'.original_state
21:     for all m ∈ N such that ⟨n, m⟩ ∈ A do
22:       temp_lock_steps ← empty sequence;
23:       temp_lock_set ← n'.lock_set;
24:       call procedure build_L(m, temp_lock_set, temp_lock_steps);
25:       {try to find a m' ∈ duplicate_state(m) with the same lock_set: }
26:       flag ← false
27:       for all node n'' ∈ N' such that n''.original_state = m do
28:         if n''.lock_set = temp_lock_set then
29:           flag ← true
30:           A' ← A' ∪ {⟨n', n''⟩}
31:           prob'(⟨n', n''⟩) ← prob(⟨n, m⟩)
32:           attach temp_lock_steps to ⟨n', n''⟩;
33:           break
34:         end if
35:       end for
36:       if flag = false then
37:         generate a new state m'
38:         m'.lock_set ← temp_lock_set;
39:         m'.original_state ← m
40:         N' ← N' ∪ {m'}
41:         if m ∈ F then
42:           F' ← F' ∪ {m'}
43:         end if
44:         cost'(m') ← cost(m)
45:         A' ← A' ∪ {⟨n', m'⟩}
```

```
46:            prob'(⟨n', m'⟩) ← prob(⟨n, m⟩)
47:            attach temp_lock_steps to ⟨n', m'⟩;
48:            enqueue(m', q)
49:          end if
50:        end for
51:      end if
52:    end while
```

Procedure build_L builds LS at m. The single member set in $L_{T'}(n')$ is passed into lock$_s$et at the calling time. LS is stored in lock$_s$et when the procedure returns. lock_steps stores the lock step sequence.

```
1: procedure build_L(m, lock_set, lock_steps);
2: for all d ∈ lock_set do
3:    call procedure remove_lock_set(m, d, lock_set, lock_steps);
4: end for
5: call procedure insert_lock_set(m, lock_set, lock_steps);
```

Procedure remove_lock_set simulates unlocking d.

```
1: procedure remove_lock_set(m, d, lock_set, lock_steps)
2: if check_unlock(m, d, lock_set) then
3:    add "unlock d" into lock_steps;
4:    lock_set ← lock_set \ {d};
5: end if
```

Function check_unlock checks unlocking conditions U1 and U2.

```
1: function check_unlock(m, d, lock_set) returns boolean
2: if d ∈ UR(m) then
3:    for all child c of d do
4:       if c ∉ lock_set then
5:          if not check_unlock(m, c, lock_set) then
6:             return false;
7:          end if
8:       end if
```

```
 9:    end for
10:    return true;
11: else
12:    return false;
13: end if
```

Procedure insert_lock_set simulating locking. It is similar to Algorithm 3.2.1.

```
 1: procedure insert_lock_set(m, lock_set, lock_steps)
 2: d ← data(m);
 3: if d ∈ lock_set then
 4:    return;
 5: end if
 6: initialize an empty stack;
 7: while d ∉ lock_set do
 8:    push d to the stack;
 9:    if d is root of the local tree then
10:       break;
11:    else
12:       d ← d.parent;
13:    end if
14: end while
15: while the stack is not empty do
16:    pop d from the stack;
17:    insert "lock d" into lock_steps;
18:    lock_set ← lock_set ∪ {d};
19:    if d.parent ≠ null then
20:       remove_lock_set(m, d.parent, lock_set, lock_steps);
21:    end if
22: end while
```

Example 3.3.5. *Given the transaction type and local lock tree in Figure 3.5, we show in Figure 3.8 the steps that Algorithm 3.3.4 generates the expanded transaction type presented in Figure 3.7. The transaction type with the lock steps inserted is in Figure 3.9.*

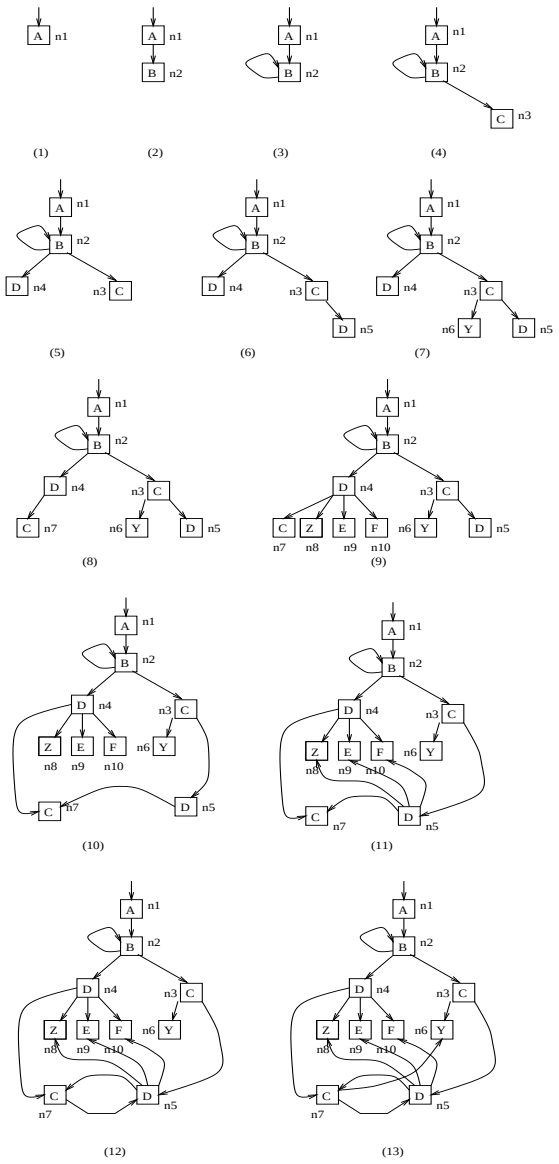


Figure 3.8: Expanding the transaction type in Figure 3.5 to the one in Figure 3.7

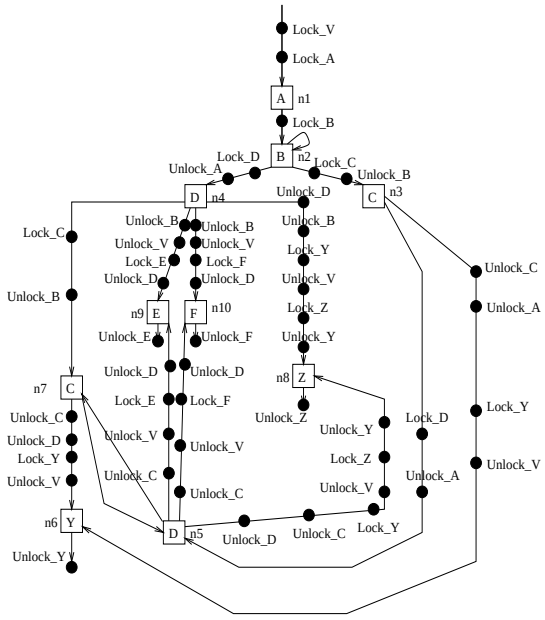


Figure 3.9: Transaction type with locking/unlocking steps (the solid cycles) inserted.

Chapter 4

Building workloads for compiled database applications

To carry on our simulated experiments to evaluate the concurrency control techniques on compiled database applications, we need a workload. There are two alternatives in getting such a workload. First, we can derive a workload from an OLTP benchmark. Second, we can choose an open source operating system as a sample of compiled database applications and map its system calls to transaction types. We introduce the two approaches separately in the chapter.

4.1 TPC-C workload

TPC-C [2] is a widely used OLTP benchmark to measure the performance of general purposed database systems. It specifies 5 transactions of a wholesale supplier, which access 9 tables that contain the information of customers, inventories, and orders. Each transaction has a probability in the mixture. As it is similar to a transaction system defined in Section 3.1, we started with TPC-C to build our workload.

4.1.1 Transforming TPC-C transactions to finite state machines

Naturally, we can build a transaction system from the TPC-C benchmark by mapping the tables to the data set and the transactions to the transaction types. So we got 9 data elements (**warehouse**, **customer**, **history**, **district**, **order**, **new_order**, **order_line**, **item**, and **stock**), and 5 transaction types (**new_order**, **payment**, **order_status**, **delivery**, and **stock_level**). From the TPC-C specifications, we also fixed the probabilities of the transaction types (for the above 5 transactions, 0.45, 0.43, 0.04, 0.04, and 0.04, respectively). The remaining work was how to build the transaction types from the TPC-C transaction programs.

We first transformed TPC-C transaction programs to finite state machines. In each program, we are only interested in embedded SQL statements and control statements (e.g. **if-else** and **for**), because they contribute to the states and transitions in the transaction system. The steps are:

1. Mapping each embedded SQL statement (one-table operation or two-table join) in the TPC-C transaction program to a component in the transaction type:
   (a) If it is a **SELECT**, **UPDATE**, **DELETE**, or **INSERT** on a single table A, we add a state n and let $data(n) = A$ (Figure 4.1(a)).
   (b) If it is a two-table join on table A and B:
      i. If the two tables are pre-selected on their keys before they are joined, we assume that the query first accesses A once, then accesses B once. So we add two nodes n_1 and n_2 such that $data(n_1) = A$ and $data(n_2) = B$, and let n_2 be the only successor of n_1 (Figure 4.1(b)).
      ii. Otherwise, we assume the query follows a nested-loop plan, and the join is transformed to Figure 4.1(c).
2. Transforming control flow in the C program to the components:
   (a) a sequential execution of embedded SQL statements $s_1, \cdots, s_n$ is transformed to a lineal path through these nodes as in Figure 4.2(a);
   (b) an **if-else** statement is transformed to a branch out in the finite state machine (Figure 4.2(b));

(c) a loop statement is transformed as Figure 4.2(c).

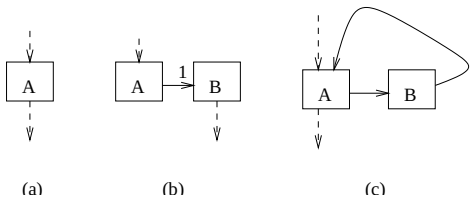


Figure 4.1: Transformations of queries

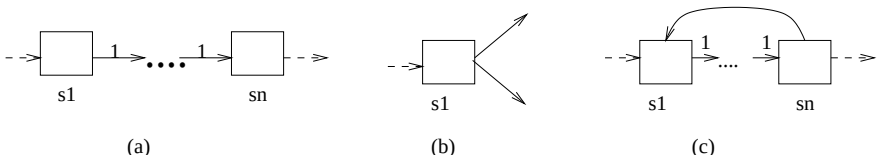


Figure 4.2: Transformations of control flows

Next, we acquired the probabilities on the transition arcs. For the join in Case 1(b)ii, the probabilities on the arcs in Figure 4.1(c) can be estimated from the table cardinality and query selectivity. For **if-else** statements (Case 2b), the probability on each branch in Figure 4.2(b) are obtainable from the TPC-C specifications. In Case 2c, if it is a **for** loop of the form,

```
for(i=0; i<n; i++){
 S1; ...; Sn;
}
```

we set the probability from s_n back to s_1 to $\frac{n-1}{n}$, and the probability to exit the loop from s_n to $\frac{1}{n}$. If it is a **while** loop, unfortunately, we have to rely on some "guess work" to approximate the probabilities.

For the simulation of 2PL, we also attached to each state with a tag **r** and **w**, indicating whether it reads or writes. If the state corresponds to a **SELECT** statement, the tag is set to **r**. If it corresponds to an **UPDATE**, **INSERT**, or **DELETE**, the tag is **w**.

So far we had the transaction types along with their probabilities to make a transaction system. The only information missing was the costs of each state, which we would acquire later on. As a byproduct, we built a global lock tree following the procedure in Section 3.3.2.

4.1.2 Partitioning the tables

Furthermore, we partitioned each table to $table_0, \cdots, table_n$, in which n is a *partition factor*. We also added indexes if it is necessary. A key-matching selection on a table first goes to the index and then goes to one of the partitions, and a fully sequential table scan goes through all the partitions one by one. We reflected such changes in the transaction types and global lock tree. If the query at a state is a key-matching search, we changed the state as in Figure 4.3(a). If it is a fully sequential table scan, we changed it according to Figure 4.3(b). For the new states that access each partition, the **r**/**w** tags are the same as the original ones. Because of the special handling of index locking in commercial database systems with 2PL, we assumed all index access operations are read and tag corresponding states with **r**.

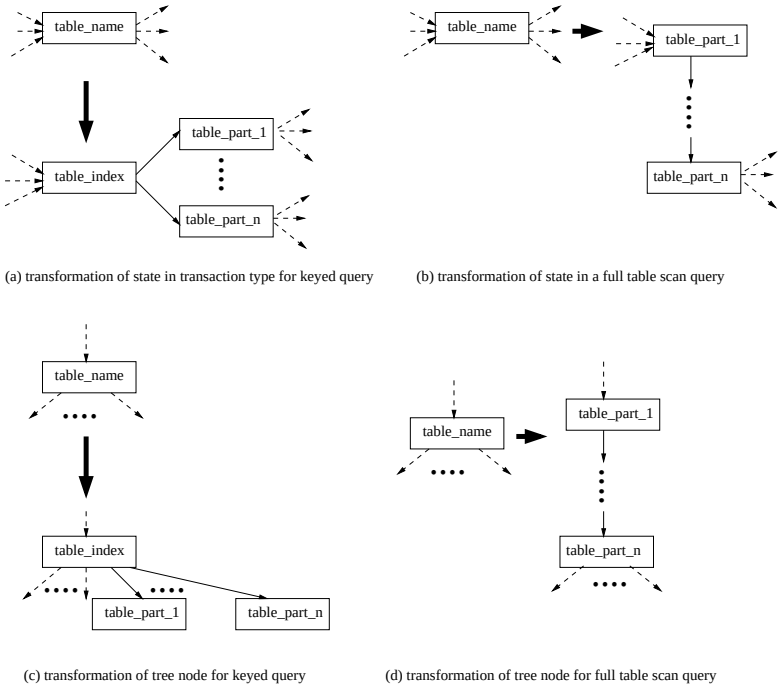
As for the global lock tree, if keyed queries on the table has a higher accumulative probability, we changed the tree node corresponding to the table as in Figure 4.3(c), otherwise we changed the node as in Figure 4.3(d)[1]. Furthermore, we built the local lock tree for each transaction type.

In our experiments, we set partition factor to 100.

4.1.3 Obtaining time costs

We also got the time costs of all operations. We created tables and indexes in an IBM DB2 database and inserted rows into the tables, following the specifications of the TPC-C benchmark. Then we ran in DB2 the SQL statements in the TPC-C programs with plugged-in parameters, and got the time costs on indexes and tables from query plan

[1]We added partitions to transaction type graphs and global lock tree after generating the original global lock tree from the transaction type graphs without partitions. The reason is that we believe it is necessary to have all partitions directly under their indexes in the lock tree.



(a) transformation of state in transaction type for keyed query

(b) transformation of state in a full table scan query

(c) transformation of tree node for keyed query

(d) transformation of tree node for full table scan query

Figure 4.3: Adding indexes and partitions to transaction types and lock trees

graphs that are provided by the DB2 visualization tool.

The TPC-C transaction types are listed in Appendix A.2.

4.2 The MINIX workload

4.2.1 Basic approach

There are some features of main-memory data processing in embedded control programs:

1. Many (although not all) lock-protected data accesses do not involve I/O at all. Their costs need to be measured in terms of CPU costs only.

2. The inter-state cost of the host program instructions cannot be ignored.

3. Overhead of locking/unlocking operations should be taken into account.

4. To reduce the locking overhead, setting the unit of lock protection to a coarse granularity may be preferable. From the main-memory database experiences [21] [9], the table-level granularity and even database-level granularity (serial execution) are good choices.

As our TPC-C workload in Section 4.1 is derived from the on-disk IBM DB2 database, it is hard to capture all the above features except the last one (by setting partition factor to 1). We find it is important to design a dedicated workload for embedded control programs to satisfy the features 1, 2, and 3. We decided to select an open source embedded control program and build our new workload from its source code. The costs at the states are to be characterized by the numbers of instructions that process the data. We also used the same measurement for inter-state transitions and locking/unlocking operations.

Open source operating systems are good candidates for our goal. The justifications are:

- The kernel data of operating systems are memory-resident.

- The semantics of the kernel data structures and system calls are well-known. Therefore, the former can be re-engineered to relational data, while the latter can be modelled as pre-defined transaction types.

- The operating systems are mainly written in C language. The C-compilers can generate the corresponding low-level assembly instructions. Therefore, we can sum up the numbers of assembly instructions to approximate the costs.

There are two candidates, *Linux* [11] [1] and *MINIX* [46]. Linux is a sophisticated operating system widely used in industry, with a rich collection of mechanism for mutual exclusion and synchronization, *e.g.*, atomic operations, spinlocks, and semaphores. However, the "hacker-written" Linux source code is too complicated for our current purpose. We decided to choose MINIX instead because of its simple and readable code and data structures, which are well introduced [46]. On the other hand, different from Linux, MINIX has no mutual exclusion and synchronization primitives except the basic interrupt disabling. When we added the locking overhead, we only considered semaphore lock, and implemented it upon MINIX source code by ourselves[2].

4.2.2 Building transaction types

First, to characterize the inter-state cost in terms of machine instructions, we change the definition of a transaction type in Definition 3.1.1.

Definition 4.2.1. *Let D be a* data set *that consists data items in the database, and each data item is identified with a name.*

A transaction type *T is defined as $T = \langle N, s, F, A, data, cost, prob \rangle$.*

- *N, s, F, A, data, and prob are the same as in Definition 3.1.1;*
- *Change the cost element in Definition 3.1.1 to: cost is a function from $N \cup A$ to nonnegative integer set. This represents the number of CPU instructions it takes on the $a \in A$ or $n \in N$.*

[2]We remind the readers that our goal is to get a workload for transaction processing in the field of embedded control programs. Therefore, we did not stick to all the features of MINIX or Linux themselves. MINIX assumes a single user and the system calls are processed sequentially. Although we got our transaction types from the MINIX code, we just use them as reference and are not restricted to such limitations of the operating system. Our workload is adjustable from fully preemptive to fully sequential, based on particular experimental settings.

We built the transaction types from the MINIX code of the system calls. At first, we did not consider the locking implementations and costs. The steps are:

1. Selecting system calls;
2. Modifying and simplifying the C code of the system calls;
3. Mapping the kernel data structures to relational tables;
4. Changing the pure C code to the C code with embedded SQL statements (C/SQL code);
5. Obtaining the finite state machines from the C/SQL code, and getting the probabilities of state transitions.
6. Generating assembly instructions from the C code at Step 3, and obtaining the costs by summing up the number of instructions;

Selecting system calls

From the MINIX code in [46], we selected 5 system calls, **fork**, **exit**, **waitpid**, **exec**, and **brk** to build up our workload. The reasons that we chose them are:

1. All these system calls are well-known and frequently used in Unix-like operating systems [49] [46].
2. They are highly related to the data structures that represent process and memory chunk descriptors.

Modifying C code

We singled out a collection of data structures of interest relating to the selected system calls. They are *process descriptors*, *opened file descriptors*, *in-memory inode descriptors*, *main-memory chunk descriptors*, and *free memory hole descriptors*. We simplified the programs that process the system calls in [46], and kept only the parts that manipulate on the above mentioned data structures.

The MINIX operating system has a microkernel architecture with 4 layers, from the lowest to the highest, *process management*, *I/O tasks*, *server processes*, and *user processes* [46]. The kernel consists of the 2 lowest layers. Many traditional kernel functionalities are moved to the 3rd layer to make the kernel small. In particular, main memory management is put to the *main memory server* (***MM***), and file system is put to the *file system server* (***FS***). Usually several layers are involved in the handling of a system call.

Example 4.2.1. *To process the system call* ***fork****,* ***MM*** *is invoked to assign the memory for the new process,* ***FS*** *is invoked to duplicate the opened file descriptors to the new process, and the low-level kernel is invoked to record the new process for further scheduling. From the code perspective,* ***MM****,* ***FS****, and the kernel all have their own* ***fork*** *handlers, each implements the corresponding processing in its own module. In addition, all the three modules have their own process descriptors and process tables. The process descriptor in the kernel keeps the register information of the process, the one in* ***MM*** *keeps its memory information, and the one in* ***FS*** *keeps its opened file information.*

For simplicity, we "flattened" the system call processing to one layer. Because we are mainly interested in processing control data in main memory, we focused on the handlers of the system calls in **MM**. We also added some processing from the kernel and **FS**. From the kernel level, we took the part that maintains the ready queue, but ignored the processing of the register values. From **FS**, we only took into account the processing of the file descriptors to reflect that the files are duplicated in **fork**, and closed in **exit** and **exec**. The code in **exec** that opens and reads the executable file to get its image, was ignored. Correspondingly, we also combined the process descriptor data structures at different levels. Although in MINIX not all of these data structures and programs are in its kernel part, here we still call them kernel data structures and kernel programs.

The kernel data structures are listed in Table 4.1. Their detailed definitions and the modified programs are in Appendix A.3.1.

Mapping data structures to relational tables

We mapped the kernel data structures defined in Section 4.2.2 to relational tables. Most of the mappings are straightforward. They are listed below, in which the left hand side is data structure in the C language, and right hand side is the table name.

description	*element data type*	*data variable*
process	**proc_t**	**proc**
in-memory inode	**inode_t**	**inode**
all opened files	**filp_t**	**filp**
files opened by a process	**filp_t**	embedded as the field **fp_filp** in **proc_t**
main-memory chunk	**mem_map_t**	embedded as the field **seg** in **proc_t**
free memory hole	**hole_t**	**hole**
ready process queue	**proc_t**	**rdy_head**, **rdy_tail**

Table 4.1: MINIX kernel data structures

- **proc**[**NR_PROCS**] → **proc**;
- **inode**[**NR_INODES**] → **inode**;
- **filp**[**NR_FILPS**] → **filp**;
- **hole**[**NR_HOLES**] → **hole**;
- **rdy_head**[**USER_Q**], **rdy_tail**[**USER_Q**] → **ready_user_proc**;
- **fp_filp**[**OPEN_MAX**] in **proc**[**NR_PROCS**] → **proc_filp**;
- **seg**[**NR_SEGS**] in **proc**[**NR_PROCS**] → **segment**.
- **1..30000** → **pids**

In the C program, segments (**seg**) are defined as a nested array inside the process descriptor (**proc_t**). Since the relational model supports only "flat" tables, we put the segment information into a separate table **segment**, and defined an artificial identifier field **seg_id** for **segment** and a corresponding foreign keys in the table **proc**. The **fp_filp** array in **proc_t** shows the $m : n$ relationship between process descriptors and file descriptors, which reflects that several files can shared the same file descriptor and a process can open several files. This cannot be encoded in relational model without an auxiliary table.

Therefore, we introduced a key **filp_id** in **filp**, and created an additional table **proc_filp** which contains the foreign keys to both **proc** and **filp**.

The DDL statements that create the tables are listed in Appendix A.3.2.

Changing the pure C code to the C code with embedded SQL statements

Next, we transferred the pure C program in Appendix A.3.1 to the corresponding C program with embedded SQL statements (*C/SQL*). The main work at the step is changing the code blocks that operate on the kernel data structures to the embedded SQL statements that manipulate the mapped tables. The C/SQL programs for the system calls are listed in Appendix A.3.3.

Obtaining finite state machines from the C/SQL code

Following the same steps as used for the TPC-workload in Section 4.1.1, we transformed the C/SQL programs to finite state machines, and obtained the probabilities of state transitions. Different from the TPC-C workload, we did not partition the table, because we assume that the unit of protection for main memory databases should be of coarse granularity, *i.e.*, the table level. The finite state machines are listed in Appendix A.3.4.

Generating assembly instructions and getting the costs

The next task was getting the costs at each transition arc and each state in the finite state machine to build up the final transaction types. Because we have mapped the statements in the C program to the embedded SQL statements, we can view the C statements as the low-level query plans for the SQL statements. Therefore, if we can calculate the cost of a "block" of C statements that represents a query plan, then we can get the costs of the states involved in the SQL query. In addition, from the C statements that do not belong to any block corresponding to a query plan, we can get the inter-state costs on the arcs.

Currently we measure the cost of a sequence of C statements with the total number of their assembly instructions. We ran the **gcc -g -S** command over the C programs. This produces the assembly code of the program without code optimization. So the sequence of assembly instructions for each C statement are generated. Then we got the costs at

transaction type	probability
fork	11.7%
exit	11.7%
waitpid	9.3%
exec	9.3%
brk	58%

Table 4.2: The probabilities of the transaction types

the states and arcs by summing up the number of instructions of their statements. When there are loops or conditional statements in the code, we approximated the numbers by *ad hoc* "averaging" or even "guessing". For more techniques details, please refer to Appendix A.3.4.

4.2.3 Getting the probabilities of the transaction types

After we have the set of transaction types, we also need their probabilities to make the final transaction system. Here we assume that there is the same percentage of **fork** and **exit** in the transaction mixture, 80% of forked processes calls **exec**, and 80% of the parent processes call **waitpid** for their child processes. **brk** is the underlying system call that implements **malloc** and **free**. We also assumed that on average each process calls **brk** for 5 times. So we calculated the percentage of each transaction type. The probabilities are given in Table 4.2.

Therefore we have finalized the whole transaction system as the workload.

4.2.4 Calculating the costs of locking

We also considered the costs of locking protections. The MINIX system has only interrupt disabling protection. This is equivalent to the serial scheduling on a uniprocessor system, and can implemented by one **cli** instruction for locking and one **sti** for unlocking. Unix/Linux also uses high level *semaphores* [49] [11] for synchronization and protection. Both locks and latches are usually implemented by semaphores. We treated exclusive locks (*x-lock*) and locks that allow sharing (*rw-lock*) separately. With the implementation of the

spinlock functions (**spin_lock** and **spin_unlock**) borrowed from the Linux code [11], we implemented a simple version of both types of locks based on MINIX data structures. We also calculated the average costs for locking and unlocking operations by totalling the numbers of machine instructions. The technical details are given in Appendix A.3.5.

Chapter 5

Experiments

5.1 Experimental setting

On the two workloads we have designed, we simulated the transaction processing under TL and the traditional 2PL, respectively, and compared their throughput. The hardware of our experiments is a workstation with an Intel P4 1.5GHz CPU, 256M RAM, and a 20G hard disk. The programs of the experiments are written in the Java language.

We programmed a discrete event simulator [24] for each locking protocol. Our experiments assume single CPU and shared memory. A queue is managed for each data item to block the transactions that are waiting for the lock on it, and a global event queue is maintained to schedule transactions that are ready to run. The TL simulator follows the locking/unlocking algorithms in Section 3.2. We ignore the runtime costs of tree traversal and TL unlocking condition testing, assuming the lock steps have already been inserted at compile time. The 2PL simulator uses strict 2PL and logs all updates. We implemented a deadlock detector for the 2PL simulator. It uses the waits-for-graph approach [8], and always aborts the youngest transaction involved in a deadlock. When a transaction is aborted, the logged updates are rolled back in reverse order. The overhead of deadlock detection is ignored. For the TPC-C workload, we also ignore the costs of locking/unlocking operations in both TL and 2PL simulators.

We model the time duration of a state as below. For each state, there is a CPU time followed by a waiting time. The waiting time is used to simulate disk I/O and

communication delaying. It follows a negative exponential distribution[1]. The time duration of a state is the sum of the CPU time and the waiting time. During the CPU time, the transaction occupies the CPU exclusively. When the waiting time starts, the transaction relinquishes the CPU, so that another transaction which is ready can run on the CPU during the waiting time of the first transaction. The ratio of the mean waiting time to the CPU time is named as *waiting factor*. For the TPC-C workload, we approximated the mean waiting time with the time cost we acquired from DB2, and set the CPU time by dividing the mean waiting time by the waiting factor. For the MINIX workload, we approximated the CPU time with the number of instructions, and set the mean waiting time by multiplying the CPU time by the logging factor. For each state that writes in 2PL simulations, we also add a logging overhead to the state cost. The logging overhead also includes a CPU time and a waiting time. The CPU time and the mean waiting time for logging are obtained by multiplying the CPU time and the mean waiting time of the original write by a *logging factor*, respectively. When a transaction is aborted, a logged write is rolled back with the CPU time and mean waiting time of the original write. For traditional on-disk databases, because the I/O overhead is high, the waiting factor is high; and because logging can be done efficiently [23], the logging factor is low. For embedded control programs that operate on main-memory data, because there is little I/O and the overhead to move data in memory is high, the waiting factor is low and the logging factor is high. By setting the two parameters waiting factor and logging factor, we are able to describe whether the operation is for on-disk or in-memory data processing.

The number of user terminals is set to 10. A transaction is submitted from a terminal after the previous one from the same terminal commits, and its transaction type is chosen based on the probabilities given by the transaction system. Transactions issued from different terminals run concurrently. In 2PL simulations, an aborted transaction is immediately restarted from the same terminal.

We generated random number sequences for transaction type selection, state transition selection, and other random events. In each *trial*, the same random number sequences

[1]Let $mean\_waiting\_time$ be the mean waiting time, $rand$ be a random number with a uniform distribution between 0 and 1, a random waiting time $waiting\_time$ is generated by the formula $waiting\_time = -log_e(rand) \times mean\_waiting\_time$.

are applied to the TL and 2PL simulations so that comparisons are made under the same condition.

5.2 Simulated results on TPC-C

For the TPC workload, we considered two versions of the 2PL, the normal 2PL that allows both shared and exclusive locks (2PL-rw), and the 2PL that allows only write operations and exclusive locks (2PL-w). We ran the TL, 2PL-rw, and 2PL-w simulators for comparison. Under different settings of logging factor and waiting factor, we conducted 30 trials, each with a simulated time of 60 minutes. The throughput of each locking protocol was measured by the average number of transactions committed per trial in its simulation. Following the TPC-C specifications, we compared the throughput w.r.t. all transaction types and the **new_order** transaction type (the one with the highest probability).

First, we set the logging factor to 0.2 and the waiting factor to 10, which simulates the on-disk database situation. The result is shown in Figure 5.1. TL performs better than 2PL-w. This is because of the logging overhead of 2PL-w. On the other hand, 2PL-rw outperforms TL by 2:1. This is because many operations in TPC-C are reads. 2PL-rw supports shared lock and allows concurrency between reads, while TL suffers from its write-only property. The logging overhead of writes has little influence on the 2PL-rw throughput. Additionally, we included the numbers of transactions that are aborted for deadlock resolution under 2PL-rw and 2PL-w in Figure 5.1. The abort rate under 2PL-w is tiny. Although there are much more aborts under 2PL-rw, the number is still relatively low compared with the large number of transactions committed. This shows that deadlock of 2PL-rw and 2PL-w does not affect their performance noticeably.

Furthermore, we switched the settings from on-disk databases to main-memory data repositories, which is the primary field of compiled database applications. We only made comparisons between TL and 2PL-rw, because by its nature the performance of 2PL-w can only get worse under the new settings. First, for main-memory data processing, the logging factor can be larger (between 1 and 10), because it is relatively cheaper to access in-memory data and more expensive to move data in main memory (logging). Therefore, we investigated how the throughput of 2PL goes down with increasing logging factors. In

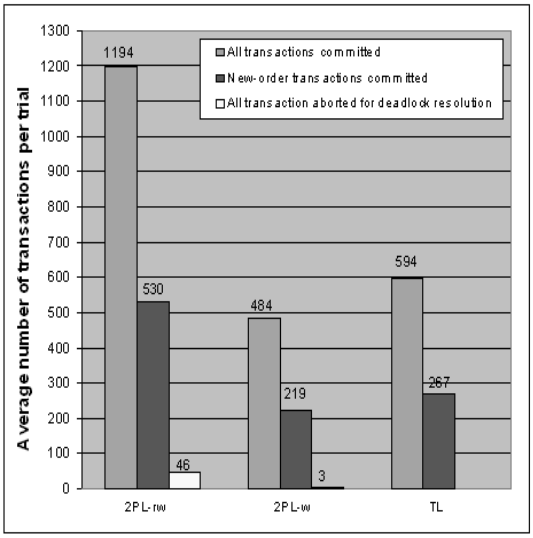


Figure 5.1: The simulated throughput comparisons between 2PL-rw, TL and 2PL-w, with logging_factor=0.2, waiting_factor=10

our experiments, we examined the throughput of 2PL-rw with logging factors 0.2, 1, 5, 15, 20, and 25, respectively. The result is presented in Figure 5.2. Although the performance of 2PL-rw decreases as the logging factor goes up, it is still better than TL until the logging factor reaches about 20, which is too high to be meaningful.

Next, we changed the waiting factor from 10 to 1, because for main-memory data processing, there is little I/O cost, and the CPU cost is dominant. We did experiments with logging factors 1, 5, 8, and 10. Figure 5.3 shows the result. With lower waiting time, the parallelism of 2PL-rw is less favored. Moreover, the throughput of 2PL-rw is reduced by aborts because of deadlock. Compared with Figure 5.2, the deadlock rate is much higher. The numbers of aborted transactions are very close under different logging factors. In particular, the **new_order** transaction type has an exceptionally high deadlock rate, so its number of commits goes down drastically. Therefore, the throughput of 2PL-rw and TL is comparable when the logging factor is between 5 and 10, a fairly reasonable range for main-memory logging.

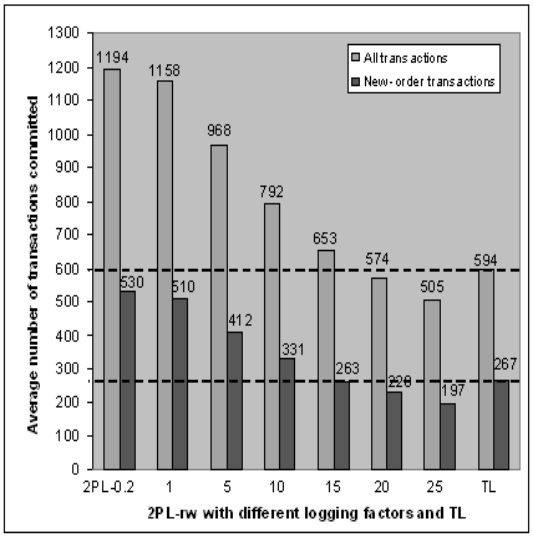


Figure 5.2: The simulated throughput comparisons between 2PL-rw with increasing logging_factor and TL, waiting_factor=10

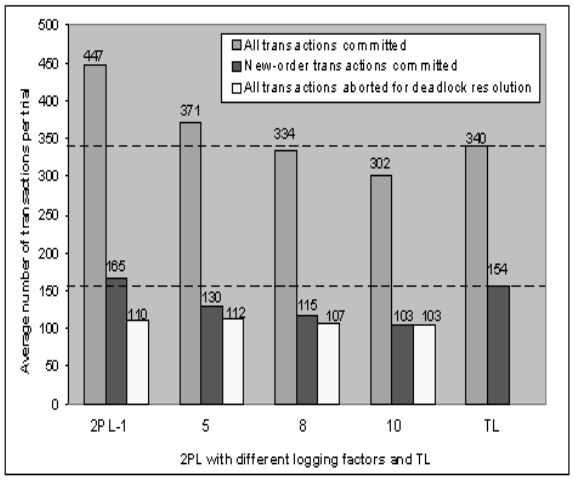


Figure 5.3: The simulated throughput comparisons between 2PL-rw with increasing logging_factor and TL, waiting_factor=1

5.3 Simulated results on the MINIX workload

We continued our experiments with the MINIX workload. We compared the throughput of TL and 2PL (with reads and writes). In our simulations, we added in the costs of inter-state transitions and the overhead of locking and unlocking. The MINIX workload is from the main-memory condition, and the costs at each state and each transition are measured by numbers of CPU instructions. For simplicity, we used CPU instruction as the time unit. We took into account the memory delay and the parallelism from the modern CPU design, and modelled them as the waiting time both of the states and of the transitions. We used the two parameters waiting factor and logging factor to acquire the waiting time and logging overhead (in 2PL).

Same as in TPC-C experiments, under the same setting of two factors, we conducted 30 trials with different random number sequences each time. The throughput is measured by the average number of transactions committed per trial. Following the idea of TPC-C, we recorded the numbers of all transaction types, and 3 selected transaction types with high probabilities, **brk**, **fork**, **exit**. We also recorded the numbers of aborts for deadlock resolutions as well. First, we simulated main memory data processing, and set the waiting factor to 1, and the logging factor to 1 and 5. The simulated time duration of the experiments was set of 3.6×10^6 time units (CPU instructions). The result is shown in Figure 5.4. We can see that the number of transaction committed TL is twice as many as 2PL even when the logging factor is 1, and the difference is far larger when it increase to 5. It is observed that 2PL always has a large number of aborts, even more than the number of commits. In fact, all transaction types except **exit** have high deadlock rates. That explains why **fork** has much lower number of commits than **exit** under 2PL in Figure 5.4 although they have the same probability in the MINIX workload (See Section 4.2.3). Under TL, the numbers of commits of **fork** and **exit** are very close.

We also simulated the on-disk data processing. We set the logging factor to 0.2, and increased waiting factor at each state to 30. We still kept the waiting fact at each transition as 1. The simulated time duration of each trial is increased to 4.32×10^7 time units. We present the result in Figure 5.5. The influence of deadlock of 2PL goes down, and the throughput of 2PL is getting closer to TL. For the transaction type **exit**, which has a low deadlock rate, 2PL has obviously more transactions commits. But for **fork** transactions

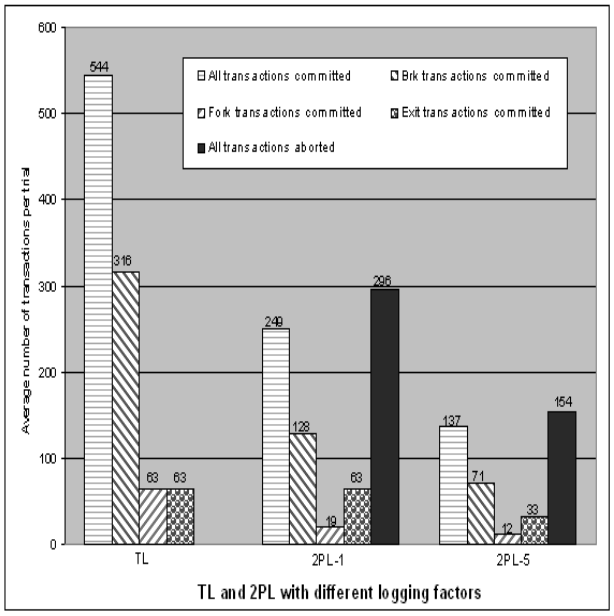


Figure 5.4: The simulated throughput comparisons between 2PL with logging factor_1 and 5 and TL, waiting_factor=1

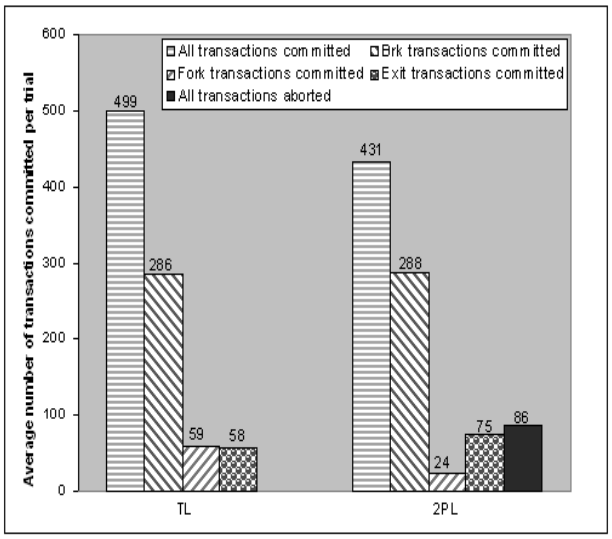


Figure 5.5: The simulated throughput comparisons between 2PL and TL, with waiting_factor=30 and logging_factor=0.2

with high likelihood of deadlock, 2PL still does poorly. For the MINIX workload, TL can in general compete with 2PL even under the on-disk setting[2]. Recall in the experiments with the TPC-C workload, 2PL does far better than TL in processing on-disk data (Figure 5.1). The reasons for the difference can be:

- Most MINIX transaction types have high deadlock rates.
- MINIX workload has a coarse granularity (table level) of lock protection, which reduces the concurrency of 2PL more than TL.

[2]We sampled some trials with the waiting factor setting to 50 and 100, and the results of comparisons remain similar.

Chapter 6

Conclusions and future work

6.1 Conclusions

In the thesis, we explored how to apply the deadlock-free TL protocol to the concurrency control of compiled database applications. Because the transaction types are predefined, we can generate all data structures, including the unlockable sets and the lock tree, that are necessary for the runtime locking/unlocking at compile time. We have designed compile-time algorithms to generate such data structures (Section 3.3.1 and 3.3.2), and the runtime TL locking/unlocking algorithms based on them (Section 3.3.3). Moreover, at compile time, we can also insert the lock steps into the transaction types at the cost of expanding the transaction types. We have also designed an algorithm for transaction type expansion and lock step insertion (Section 3.3.3).

We evaluated the performance of TL using the simulation method. We have built two compiled database application workloads for our simulations. The first is from the OLTP benchmark TPC-C, and the second is from the open source operating system MINIX. We transformed the programs into transaction types. For the TPC-C workload, we got the time cost by running the SQL queries in DB2. For the MINIX one, we tried to approximate the features of main-memory data processing, and measured the costs by CPU instructions.

Our simulated experiments compared the throughput of the TL protocol with the classical 2PL protocol. By changing parameters, we did the simulations under both on-disk and and main-memory settings. For on-disk setting, TL is in general inferior to 2PL be-

cause it has no shared lock. Although TL has no logging overhead, it has obvious effects on the throughput comparison only when the transactions are write-only or dominated by writes. On the other hand, for main-memory data processing, which has low I/O costs and high 2PL logging costs, TL performs comparably to 2PL. In particular, for the workload which has a high deadlock rate under 2PL, the deadlock-free TL has a clear lead.

Our work can be added into the query optimization approaches in [13] and [47]. With the generated query plans as the transaction types, the extended optimizer further builds the lock tree, generates the locking/unlocking code that implements the algorithms in Section 3.2, and inserts it into the generated query processing code. As an alternative, the optimizer can also modify the transaction types and then insert the lock steps directly into the transaction types. Since TL has no deadlock, no logging and recovery are necessary. Therefore, it becomes entirely possible to decouple concurrency control from from any recovery considerations in compiled database applications.

6.2 Future work

As the MINIX workload is still very primitive, we need to improve our compiled database workload in the future. There are several directions to consider:

- the SMP environment under which spinlock plays an important role;
- a workload from a more sophisticated embedded control program, *e.g.*, Linix;
- an automated tool that extracts the workload from the source code of embedded control programs.

In the thesis, a transaction type is modelled a probabilistic finite state machine. All compile-time and runtime processing are based on it. We need to further formalize how to map between the finite state machine and the query plan in terms of the plan algebra [13]. This is more important for the insertion of lock steps. As we usually have to expand the transaction type, we need to modify the corresponding query plan as well to make it practicable.

We also plan to consider the following:

- an in-depth study of the tree fitness, and
- an exploration on how to extend the TL-based concurrency control to the multilevel transaction model [52] [53] (See Section 2.3.2 and 2.3.3).

Appendix

A.1 Case study: abort handling in Linux system call fork

We take Linux kernel source code [1] as a study case of low-level embedded control program, and view its system calls as the predefined transaction types. An operation can be either a block of statements or a function call. Usually, a system call can not be "aborted" by the caller. But it may still end up with a failure, *e.g.*, when there is no enough memory or other resources. This is usually indicated by an error code returned to the caller. And the implementation of a system call must cleanup the partial changes to the kernel data structures if the system call fails, and thus provides a view of "atomicity" for the caller. We picked up an important system call **fork**[1] and inspected the program code[2] that implements it.

In the source code, we find not all updates on the kernel data need to be recovered at the failure. More important, for those operations need to be recovered, compensation operations are supplied. An operation and its compensation are typically implemented with a pair of functions with straightforward names, *e.g.*, **copy_fs** and **exit_fs**. When a failure is encountered, the program executes the compensation operations for those already finished operations in a reverse order. Thus the atomicity of the system called is guaranteed. The program control flow of **fork**, including operations and their compensations, is shown

[1]For a more detailed introduction of **fork**, please refer to [49] (under Unix) and [11] (under Linux).

[2]The source code is at "http://lxr.linux.no/source/kernel/fork.c". And the main function is **do_fork**.

in Figure A.1. We find this paradigm follows the two-level saga transaction model [20] (surveyed in Section 2.3.4).

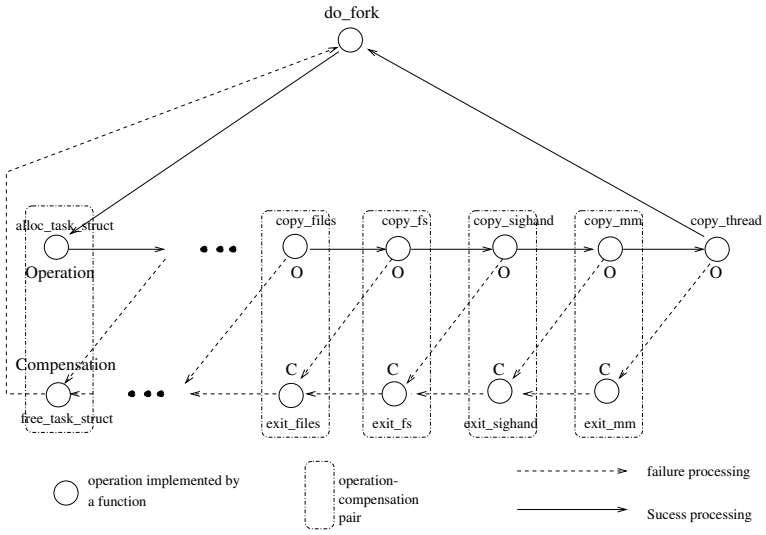


Figure A.1: The control flow of **fork**

A.2 The TPC-C workload

The TPC-C transaction types are shown in Figure A.2 to A.6. The number attached to each state is its cost, and the text inside a state box is the name of the data item. The **r** and **w** tags indicate whether a state reads or writes. The number on each arc is the probability for the transition, and its values is between 0 and 1. The partition factor is set to be 1 for simplicity.

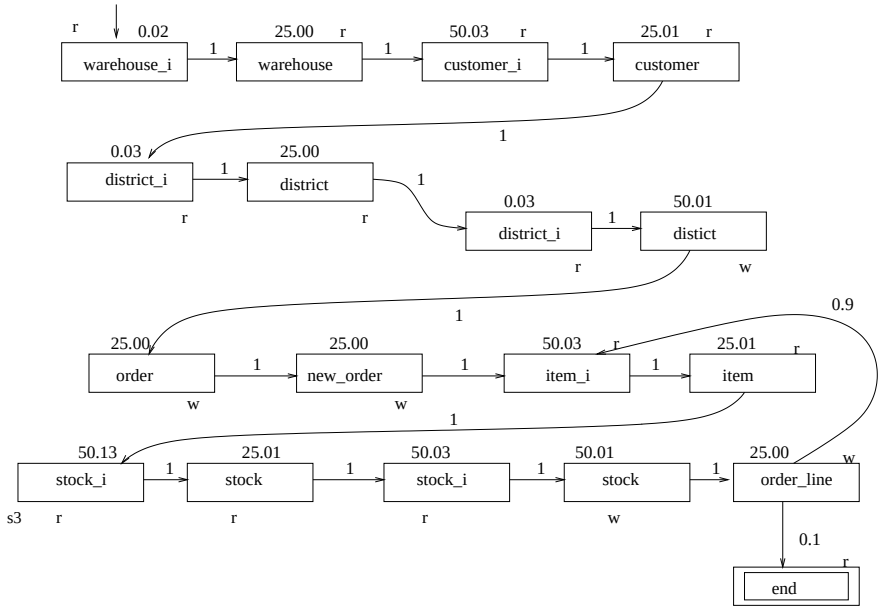


Figure A.2: The **new_order** transaction type

A.3 The MINIX workload

A.3.1 C programs that handle the selected system calls

In the section, we list the programs of the selected system calls. They are modified from the MINIX source code in [46]. In the programs for **fork** and **exec**, the function **phys_copy** called is a function written in assembly language in MINIX source code.

Constants, types, and data structures

The constants and types are taken from various header files in [46]. We gather their definitions in file **types.h**.

```
/********************************************************/
/*            constants                                 */
/********************************************************/
/* error messages */
#define OK   0
#define EGENERIC (-99) /* generic errors */
#define ECHILD (-10)   /* no child process */
```

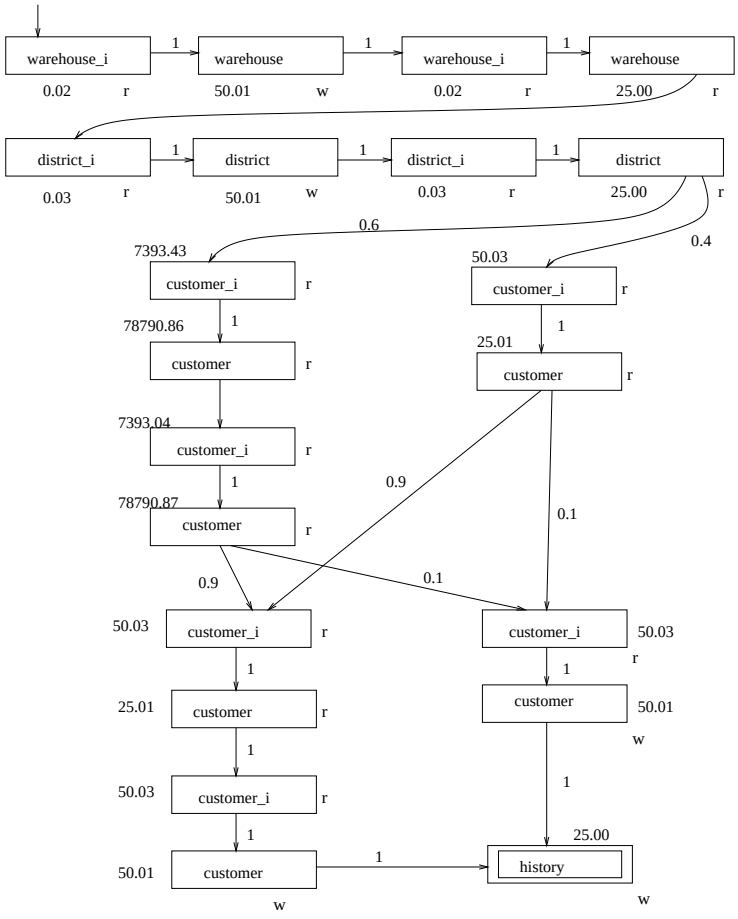


Figure A.3: The **payment** transaction type

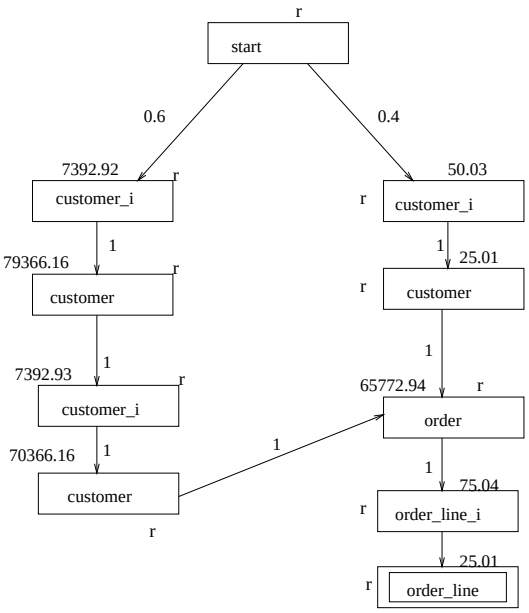


Figure A.4: The **order_status** transaction type

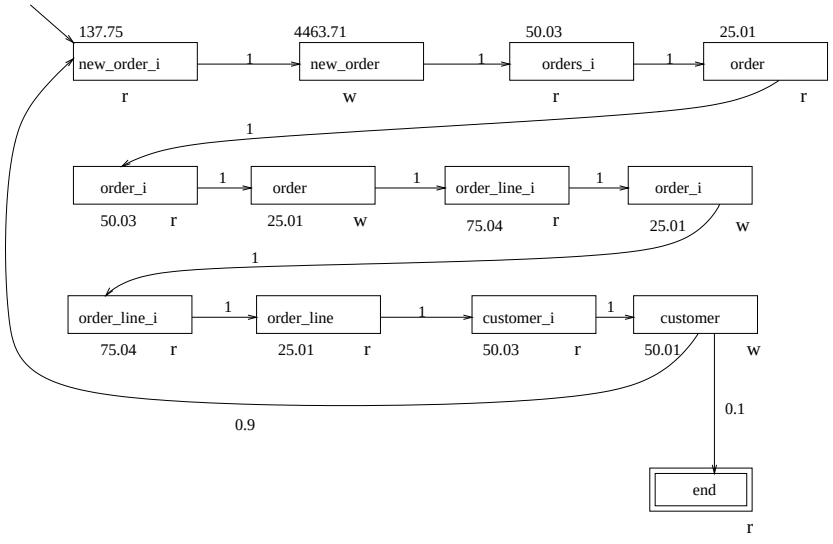


Figure A.5: The **delivery** transaction type

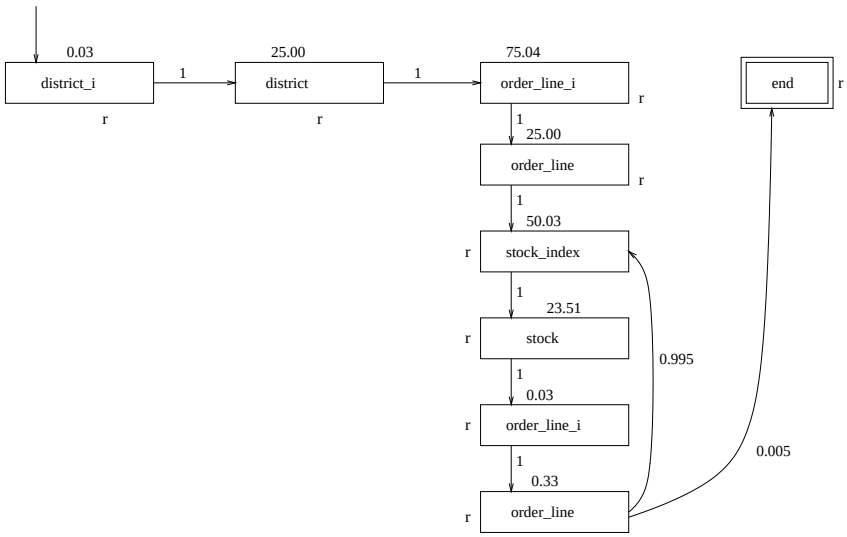


Figure A.6: The **stock_level** transaction type

```
#define EAGAIN (-11)   /* resource temporarily unavailable */
#define ENOMEM (-12)   /* no enough memory */

/* memory chunk */
#define CLICK_SIZE 256 /* click unit size */
#define CLICK_SHIFT 8  /* bit shift */

/* segments */
#define NR_SEGS 3   /* segments text, data, stack */
#define T       0   /* text */
#define D       1   /* data */
#define S       2   /* stack */

/* holes */
#define NR_HOLES 128     /* line 18820 */

/* file system */
#define OPEN_MAX 20      /* line 167 */
#define NR_FILPS 128     /* line 19506 */
#define NR_INODES 65     /* line 19507 */

/* processes */
#define MM_PROC_NR 0   /* process id of memory manager */
#define INIT_PID 1     /* process id of init process */
#define IDLE     2     /* idle task */
```

```
#define NR_PROCS 32    /* maximum number of user processes */
#define IN_USE 001     /* the process array element is used */

#define WAITING 002   /* the process is waiting: it has called waitpid
                         and is waiting a child to exit */
#define HANGING 004   /* the process is hanging: it has called exit and
                         is waiting for the parent to call waitpid */
#define SEPARATE 040  /* the process has separate text and data segments */

/* misc constants */
#define FALSE 0
#define TRUE  1

#define MAX(a, b) ((a) > (b) ? (a):(b));
#define MIN(a, b) ((a) < (b) ? (a):(b));

#define ARG_MAX  4096  /* args + environ on small
                          machines (for exec()) */
/*********************************************************/
/*            types and structures                       */
/*********************************************************/
typedef int boolean;
typedef int pid_t;             /* process id */
typedef long time_t;   /* time */
typedef unsigned short mode_t; /* file type and permission */
typedef unsighed long off_t;   /* position in a file for r/w */
typedef short dev_t;           /* device id */
typedef unsigned short ino_t;  /* inode id */
typedef unsigned reg_t;        /* register value format */

typedef unsigned int vir_clicks; /* virtual memory chunk */
typedef unsigned int vir_bytes;  /* virtual memory byte */
typedef unsigned int phys_clicks; /* physical memory chunk */
typedef unsigned int phys_bytes; /* physical memory byte */

/* memory segment */
struct mem_map {
  vir_clicks mem_vir;   /* virtual address */
  phys_clicks mem_phys; /* physical address */
  vir_clicks mem_len;   /* length in clicks */
};

/* holes */
struct hole_t {
  phys_clicks h_base;    /* physical starting address */
  phys_clicks h_len;     /* length */
  struct hole_t *h_next; /* pointer to the next entry */
```

```
} hole[NR_HOLES];

#define NIL_HOLE (struct hole_t *) 0  /* null hole pointer */

struct hole_t *hole_head;   /* pointer to the first hole */
struct hole_t *free_slots;  /* pointer to the first
                               unused hole table slot */

/* inode */
struct inode_t {
  int    count;             /* reference count:
                               if it is 0, the entry is free */
  time_t i_ctime;           /* when was inode itself changed*/
  dev_t i_dev;              /* which device is the inode on */
  ino_t i_num;              /* inode number on its device */
} inode[NR_INODES];

/* file descriptor */
struct filp_t {
  int filp_count;               /* how many processes share this slot?*/
  struct inode_t *filp_ino;     /* pointer to the inode */
  mode_t mode;                  /* file mode and permission */
  off_t  pos;                   /* position for read or write */
} filp[NR_FILPS];


#define NIL_FILP (struct filp_t *)0  /* null filp pointer */

/* process descriptor */
struct proc_t {
  pid_t  pid;          /* process id */
  pid_t  parent_pid;   /* parent process id */
  pid_t  wpid;         /* pid this process is waiting for */

  char   exitstatus;   /* storage for status when process
                          exits                          */
  char   sigstatus;    /* storage for signal number for
                          killed processes               */

  struct mem_map seg[NR_SEGS]; /* text, data, stack segments */

  /* File identification for sharing */
  ino_t   ino;    /* inode number of exec file */
  dev_t   dev;    /* device number of file system */
  time_t  ctime;  /* inode changed time */

  unsigned proc_flags; /* flag bits */
  reg_t sp;       /* stack pointer */
```

```
  struct filp_t *fp_filp[OPEN_MAX]; /* opened file descriptors */
  struct proc_t *p_next_ready;  /* pointer to the next process
                                   in the ready queue          */
} proc[NR_PROCS];

/* ready queues */
#define TASK_Q   0   /* kernel tasks */
#define SERVER_Q 1   /* servers */
#define USER_Q   2   /* user process */
#define NQ 3         /* totally 3 queues */

struct proc_t *rdy_head[NQ]; /* header of a ready queue */
struct proc_t *rdy_tail[NQ]; /* tail of a ready queue */

struct proc_t *proc_ptr; /* current running process */
struct proc_t *bill_ptr; /* process to be billed */
```

Memory allocation

The function **alloc_mem** allocates a memory chunk from the free memory holes to a process as one of its segments, and **free_mem** frees the memory chunk of a segment, and returns it to the hole list. **del_slot** is an auxiliary function to delete a slot in the hole list. They are modified from the functions of the same names in [46].

```
#include "types.h"

phys_clicks alloc_mem(phys_clicks clicks)
{
  register struct hole_t *hp, *prev_ptr;
  phys_clicks old_base;

  hp = hole_head;

  /* scan the hole list */
  while(hp != NIL_HOLE) {
    if(hp->h_len >= clicks) {
      /* we find a hole big enough, use it */
      old_base = hp->h_base;
      hp->h_base += clicks;
      hp->h_len -= clicks;

      if(hp->h_len == 0){
        del_slot(prev_ptr, hp);
      }
      return(old_base);
```

```
    }
    prev_ptr = hp;
    hp = hp->h_next;
  }
  return(NO_MEM);
}

void free_mem(phys_clicks base,
              phys_clicks clicks)
{
  boolean flag;
  struct hole_t *hp, *hp1, *prev_hp1, *hp2,
                *new_ptr, *prev_ptr;

  if(clicks == 0) return;

  new_ptr = free_slots;

  /* trying add the freed memory to front of a hole */
  flag = FALSE;
  hp1 = (prev_hp1 = hole_head);
  while(hp1 != NIL_HOLE){
    if(hp1->h_base == base + clicks){
      flag = TRUE;
      hp1->h_base = base;
      hp1->h_len  = hp1->h_len + clicks;
      break;
    }
    prev_hp1 = hp1;
    hp1 = hp1->h_next;
  }

  /* trying add the freed memory to the end of
     a hole */
  hp2 = hole_head;
  while(hp2 != NIL_HOLE){
    if(hp2->h_base + hp2->h_len == base){
      if(!flag){
        flag = TRUE;
        hp2->h_len += clicks;
      }
      else{
        /* we need to merge the two holes */
        hp2->h_len += hp1->h_len;
        del_slot(prev_hp1, hp1);
      }
      break;
```

```
    }
    hp2 = hp2->h_next;
  }

  /* we cannot attach the memory chunk to
     any existing hole. So we have to add
     a new hole to the hole list.         */
  if(!flag){
    new_ptr->h_base = base;
    new_ptr->h_len  = clicks;
    free_slots = new_ptr->h_next;
    hp = hole_head;

    if(hp != NIL_HOLE || base <= hp->h_base) {
      new_ptr->h_next = hp;
      hole_head = new_ptr;
    }
    else{
      while(hp != NIL_HOLE && hp->h_base < base){
        prev_ptr = hp;
        hp = hp->h_next;
      }
      new_ptr->h_next = prev_ptr->h_next;
      prev_ptr->h_next = new_ptr;
    }
  }
}

void del_slot(register struct hole_t prev_ptr,
              register struct hole_t hp)
{
  if(hp==hole_head)
    hole_head = hp->h_next;
  else
    prev_ptr->h_next - hp->h_next;
  hp->h_next = free_slots;
  free_slots = hp;
}
```

fork

The function is combined and modified from the functions **do_fork** in **MM**, **do_fork** in **FS**, and **do_newmap** in the kernel part in [46].

```
#include <string.h>
#include "types.h"
```

```
int fork(pid_t parent_pid){
  int i, j;
  int index;
  pid_t pid;
  boolean flag;
  int proc_in_use;
  struct proc_t parent_proc;
  phys_clicks prog_clicks, child_base = 0;
  phys_bytes  prog_bytes, parent_abs, child_abs;

  ino_t ino;
  dev_t dev;
  time_t ctime;
  unsigned proc_flags;
  pid_t wpid;
  reg_t sp;
  struct mem_map text_seg, data_seg, stack_seg;

  /* count the number of processes */
  proc_in_use = 0;
  for(i=0; i<NR_PROCS; i++){
    if(proc[i].proc_flags & IN_USE != 0) {
      proc_in_use++;
    }
  }

  /* if the number of processes exceeds the
     system limit, return error. */
  if (proc_in_use > NR_PROCS) {
    return(EAGAIN);
  }

  /* get the information of the parent process. */
  for(i=0; i<NR_PROCS; i++){
    if((proc[i].proc_flags & IN_USE != 0) &&
       (proc[i].pid == parent_pid))
    {
      parent_proc = proc[i];
      break;
    }
  }
  ino = parent_proc.ino;
  dev = parent_proc.dev;
  ctime = parent_proc.ctime;
  sp = parent_proc.sp;
  wpid = parent_proc.wpid;
  proc_flags = parent_proc.proc_flags;
```

```
/* calculate the memory that is needed for the new
   process. */
/* stack */
stack_seg.mem_vir = parent_proc.seg[S].mem_vir;
stack_seg.mem_len = parent_proc.seg[S].mem_len;

/* data */
data_seg.mem_vir = parent_proc.seg[D].mem_vir;
data_seg.mem_len = parent_proc.seg[D].mem_len;

/* text */
text_seg.mem_len = parent_proc.seg[T].mem_len;
text_seg.mem_vir = parent_proc.seg[T].mem_vir;

/* total memory size for the new process */
prog_clicks = (phys_clicks)parent_proc.seg[S].mem_len;
prog_clicks += (parent_proc.seg[S].mem_vir - parent_proc.seg[D].mem_vir);

/* allocate memory */
if((child_base = alloc_mem(prog_clicks))==NO_MEM)return(EAGAIN);

/* Create a copy of the parent's core image for the child */
child_abs = (phys_bytes)child_base << CLICK_SHIFT;
parent_abs = (phys_bytes)parent_proc.seg[D].mem_phys << CLICK_SHIFT;

phys_copy(parent_abs, child_abs, prog_bytes);

/* now get an id for the new process */
for(i=0; i<30000; i++){
  flag = FALSE;
  for(j=0; j< NR_PROCS; j++) {
    if((proc[j].pid == i) && (proc[j].proc_flags & IN_USE != 0)){
      flag = TRUE;
      break;
    }
  }
  if(!flag) {
    pid = i;
    break;
  }
}

/* set the text segment physical address */
if(!(proc_flags & SEPARATE)){
  text_seg.mem_phys = child_base;
}
else {
```

```
    text_seg.mem_phys = parent_proc.seg[T].mem_phys;
  }

  /* find an available entry in the . */
  for(i=0; i<NR_PROCS; i++){
    if(0 == (proc[i].proc_flags & IN_USE)){
      index = i;
      break;
    }
  }

  /* there must be one. assign the values to the
     fields */
  proc[index].pid = pid;
  proc[index].parent_pid = parent_pid;
  proc[index].ino = ino;
  proc[index].dev = dev;
  proc[index].ctime = ctime;
  proc[index].sp = sp;
  proc[index].seg[T] = text_seg;
  proc[index].seg[D] = data_seg;
  proc[index].wpid = wpid;
  proc[index].proc_flags = proc_flags;

  /* share the open file descriptors with the
     new child */
  for(i=0; i<OPEN_MAX; i++){
    proc[index].fp_filp[i] = parent_proc.fp_filp[i];

    if(proc[index].fp_filp[i] != NIL_FILP){
      (proc[index].fp_filp[i])->filp_count++;
    }
  }

  /* add to ready queue *
  if(rdy_head[USER_Q] == NIL_PROC)
    rdy_tail[USER_Q] = &proc[index];
  rdy_head[USER_Q] -> p_next_ready = rdy_head[USER_Q];
  rdy_head[USER_Q] = &proc[index];

  return (pid);
}
```

exit

The function is combined and modified from the handlers **do_exit** in **MM**, and **do_xit** in the kernel part in [46].

```
#include "types.h"

void do_exit(pid_t pid){
  int i, j;
  struct proc_t *current_proc_ptr, *init_proc_ptr,
         *parent_proc_ptr;
  struct proc_t *xp, *rp, **qtail;
  boolean text_shared, right_child;
  unsigned parent_waiting;

  pid_t parent_pid;
  pid_t wait_pid, parent_wait_pid;
  char exit_status;
  char sig_status;
  unsigned proc_flags, parent_proc_flags;
  ino_t ino;
  dev_t dev;
  time_t ctime;

  /* find the index of the process to exit. */
  for(i=0; i<NR_PROCS; i++){
     if((proc[i].proc_flags & IN_USE != 0) && (proc[i].pid == pid)){
       /* there must be one, because myself exists */
       current_proc_ptr = &proc[i];
       break;
     }
  }

  /* first close all open file. */
  for(i=0; i<OPEN_MAX; i++){
    if((current_proc_ptr->fp_filp[i])!=NIL_FILP){
      (current_proc_ptr->fp_filp[i])->filp_count--;

      if((current_proc_ptr->fp_filp[i])->filp_count == 0){
        (current_proc_ptr->fp_filp[i])->filp_ino->count--;
      }
      current_proc_ptr->fp_filp[i] = NIL_FILP;
    }
  }

  /* get all information of the current process. */
  parent_pid  = current_proc_ptr->parent_pid;
```

```
wait_pid    = current_proc_ptr->wpid;
exit_status = current_proc_ptr->exitstatus;
sig_status  = current_proc_ptr->sigstatus;
proc_flags    = current_proc_ptr->proc_flags;
ino         = current_proc_ptr->ino;
dev         = current_proc_ptr->dev;
ctime       = current_proc_ptr->ctime;

/* remove the process from the ready queue,
   and pick the next process to run. */
if((xp = rdy_head[USER_Q]) == current_proc_ptr){
  rdy_head[USER_Q] = xp->p_next_ready;

  /* pick the next proc to run */
  if ( (rp = rdy_head[TASK_Q]) != NIL_PROC) {
      proc_ptr = rp;

  }
  else
  if ( (rp = rdy_head[SERVER_Q]) != NIL_PROC) {
     proc_ptr = rp;
  }
  else
  if ( (rp = rdy_head[USER_Q]) != NIL_PROC) {
      proc_ptr = rp;
      bill_ptr = rp;

  }else{
  /* No one is ready.  Run the idle task.  The idle task might be made an
   * always-ready user task to avoid this special case.
   */
    bill_ptr = proc_ptr = &proc[IDLE];
  }
}
qtail = &rdy_tail[USER_Q];

while (xp->p_next_ready != current_proc_ptr){
  if ( (xp = xp->p_next_ready) == NIL_PROC) break;
}

if(xp != NIL_PROC){
  xp->p_next_ready = xp->p_next_ready->p_next_ready;
  if (*qtail == current_proc_ptr) *qtail = xp;
}

/* find whether the text segment is shared */
text_shared = FALSE;
for(i=0; i<NR_PROCS; i++) {
```

```
  if((proc[i].proc_flags & (IN_USE | SEPARATE | HANGING))
     != (IN_USE | SEPARATE)) continue;
  if( proc[i].pid == pid || proc[i].ino != ino
     || proc[i].dev != dev || proc[i].ctime != ctime)
    continue;
  text_shared = TRUE;
  break;
}

/* if text is not shared, free the text segment */
if(!text_shared){
  free_mem((current_proc_ptr->seg[T]).mem_phys,
           (current_proc_ptr->seg[T]).mem_len);
}

/* free the data and stack segments */
free_mem((current_proc_ptr->seg[D]).mem_phys,
         (current_proc_ptr->seg[S]).mem_len +
         (current_proc_ptr->seg[S]).mem_vir
         - (current_proc_ptr->seg[D]).mem_vir);

/* find the init process, and let it be parent of the
   children processes if necessary. */
for(i=0; i<NR_PROCS; i++){
  if((proc[i].proc_flags & IN_USE) || (proc[i].pid == INIT_PID)){
    init_proc_ptr = &proc[i];
    break;
  }
}
parent_waiting = init_proc_ptr->proc_flags & WAITING;

/* if parent is waiting */
if(parent_waiting){
  for(i=0; i<NR_PROCS; i++){
    if((proc[i].proc_flags & IN_USE) &&
      (proc[i].parent_pid == pid)){
      /* if the child already exits */
      if(proc[i].proc_flags & HANGING){
        /* free the child entry. */
        proc[i].proc_flags = 0;
        init_proc_ptr->proc_flags &= ~WAITING;
        break;
      }
    }
  }
}

/* reset the parent of the children to
```

```
    the process init */
  for(i=0; i<NR_PROCS; i++){
    if((proc[i].proc_flags & IN_USE) &&
       (proc[i].parent_pid == pid))
    {
      proc[i].parent_pid == INIT_PID;
    }
  }

  /* find the parent of the exiting process */
  for(i=0; i<NR_PROCS; i++){
    if((proc[i].proc_flags & IN_USE) &&
       (proc[i].pid = parent_pid)){
      parent_proc_ptr = &proc[i];
      break;
    }
  }
  parent_wait_pid = parent_proc_ptr->wpid;
  parent_wait_pid = parent_proc_ptr->proc_flags & WAITING;

  /* Is the parent waiting for the exiting process? */
  if(parent_wait_pid == -1 || parent_wait_pid == pid){
    right_child = TRUE;
  }
  else{
    right_child = FALSE;
  }

  if(parent_waiting && right_child){
    /* the parent is waiting for the exiting
       process, so free the process descriptor,
       and clear the waiting flag of the parent.
    */
    current_proc_ptr->proc_flags = 0;
    parent_proc_ptr->proc_flags &= ~WAITING;
  }
  else{
    /* set the flag showing the exiting process
       is a zombie.  */
    current_proc_ptr->proc_flags |= HANGING;
  }
}
```

waitpid

The code is modified from the **do_waitpid** function in **MM** in [46].

```
#include "types.h"
int waitpid(pid_t current_pid,
            pid_t pidarg)
{
  int i;
  unsigned parent_proc_flags;
  struct  proc_t  *current_proc, *child_proc;
  pid_t   wait_pid;
  boolean flag;

  /* pidarg == -1, wait for any child,
     pidarg > 0, wait for a special child with the pid specified by
     the parameter.
  */

  if(pidarg != -1 && pidarg <= 0) return(EGENERIC);

  /* get the info of current process. */
  for(i=0; i<NR_PROCS; i++){
    if(proc[i].pid == current_pid){
      current_proc = &proc[i];
      break;
    }
  }

  if(pidarg == -1){
    /* pidarg == -1: waiting for any child. */
    flag = FALSE;
    /* scan the children to see if there is
       a zombie. */
    for(i=0; i<NR_PROCS; i++){
      if((proc[i].proc_flags & IN_USE)
         && proc[i].parent_pid == current_pid){
        child_proc = &proc[i];
        flag = TRUE;
        if(child_proc->proc_flags & HANGING){
          /* find a zombie, delete the process descriptor,
             and clear the parent process flag. */
          child_proc->proc_flags = 0;
          current_proc->proc_flags &= ~WAITING;
          return(OK);
        }
      }
    }
    if(!flag){
      /* Error: we have no child */
      return (ECHILD);
    }
```

```
  else{
   /* pidarg is a certain pid
      that we are waiting for */
    flag = FALSE;
    for(i=0; i<NR_PROCS; i++){
      if(proc[i].proc_flags & IN_USE) &&
         proc[i].parent_pid == current_pid &&
         proc[i].pid == pidarg){
        child_proc = &proc[i];
        flag = TRUE;
        if(child_proc->proc_flags & HANGING){
          /* find a zombie, delete the process
             descriptor, and clear the parent
             process flag */
          child_proc->proc_flags = 0;
          current_proc->proc_flags &= ~WAITING;
          return(OK);
        }
      }
    }
    /* no child, return an error. */
    if(!flag){
      /* no qualified children, must be error */
      return(ECHILD);
    }
  }
  current_proc->proc_flags |= WAITING;
  return (OK);
}
```

exec

The code is combined and modified from the functions with the same name **do_exit** from **MM** and **FS** in [46]. As a simplification, the code here omits the procedure that reads in the executable file from the file system. We assume the image of executable is already in the main memory. The information of the image is passed to the system call handler as parameters.

```
#include "types.h"
#include <string.h>

int exec(pid_t current_pid,
         vir_bytes stack_ptr,
         vir_bytes stk_bytes,
         vir_bytes data_bytes,
         vir_bytes bss_bytes,
```

```
        vir_bytes text_bytes,
        vir_bytes tot_bytes,
        ino_t     ino,
        dev_t     dev,
        time_t    ctime,
        int       ft
       )
{
  int i;
  struct proc_t *current_proc_ptr, *mm_proc_ptr;
  vir_bytes src, dst;
  vir_bytes data_vir_addr, data_phy_addr;
  vir_bytes bytes, base, bss_offset, count;
  vir_clicks max_hole_clicks, new_base, proc_clicks;
  boolean text_shared, delete_text;
  vir_clicks text_clicks, data_clicks, gap_clicks, stack_clicks;
  phys_clicks tot_clicks;
  struct hole_t *hp;
  struct mem_map text_seg;
  ino_t old_ino;
  dev_t old_dev;
  time_t old_ctime;
  static char zero[1024];
  static char mbuf[ARG_MAX];

  /* find the process descriptor of the current process */
  for(i=0; i<NR_PROCS; i++){
    if((proc[i].proc_flags & IN_USE != 0)
       && (proc[i].pid == current_pid)){
      current_proc_ptr = &proc[i];
      old_ino = current_proc_ptr->ino;
      old_dev = current_proc_ptr->dev;
      old_ctime = current_proc_ptr->ctime;
      break;
    }
  }

  /* store the stacks of the process temporarily at the memory server. */
  data_vir_addr = (current_proc_ptr->seg[D]).mem_vir << CLICK_SHIFT;
  data_phy_addr = (current_proc_ptr->seg[D]).mem_phys << CLICK_SHIFT;
  src = stack_ptr - data_vir_addr + data_phy_addr;

  /* find the memory management server. */
  for(i=0; i<NR_PROCS; i++){
    if((proc[i].proc_flags & IN_USE != 0)
       && (proc[i].pid == MM_PROC_NR)){
      mm_proc_ptr = &proc[i];
      break;
```

```
  }
}

data_vir_addr = (mm_proc_ptr->seg[D]).mem_vir << CLICK_SHIFT;
data_phy_addr = (mm_proc_ptr->seg[D]).mem_phys << CLICK_SHIFT;
dst = (vir_bytes)mbuf - data_vir_addr + data_phy_addr;

phys_copy(src, dst, stk_bytes);

/* find whether the new executable image is already used by
   some existing process and whether we can still share it */
text_shared = FALSE;
for(i=0; i<NR_PROCS; i++) {
  if((proc[i].proc_flags & (IN_USE | SEPARATE | HANGING))
     != (IN_USE | SEPARATE)) continue;
  if( proc[i].pid == current_pid || proc[i].ino != ino
     || proc[i].dev != dev || proc[i].ctime != ctime)
    continue;
  text_shared = TRUE;
  text_bytes = 0;
  text_seg = proc[i].seg[T];  /* shared text */
  break;
}

/* sizes of the segments for the executed proc */
text_clicks = ((unsigned long) text_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
data_clicks = (data_bytes + bss_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
stack_clicks = (stk_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
tot_clicks = (tot_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
gap_clicks = tot_clicks - data_clicks - stack_clicks;
if ((int)gap_clicks < 0) return(ENOMEM);

/* get the hole of maximum size to assign memory for the
   segments. */
max_hole_clicks = 0;
hp = hole_head;
while(hp != NIL_HOLE) {
  if(hp->h_len > max_hole_clicks) max_hole_clicks = hp->h_len;
  hp = hp->h_next;
}
if(max_hole_clicks < tot_clicks + text_clicks) return (EAGAIN);

/* if the text is shared by other process, we cannot delete it. */
delete_text = TRUE;
for(i=0; i<NR_PROCS; i++) {
  if((proc[i].proc_flags & (IN_USE | SEPARATE | HANGING))
     != (IN_USE | SEPARATE)) continue;
  if( proc[i].pid == current_pid || proc[i].ino != old_ino
```

```
     || proc[i].dev != old_dev || proc[i].ctime != old_ctime)
    continue;
  delete_text = FALSE;  /* it is shared */
  break;
}

/* delete the segments of the original process. */
if(delete_text){
  free_mem((current_proc_ptr->seg[T]).mem_phys,
           (current_proc_ptr->seg[T]).mem_len);
}
free_mem((current_proc_ptr->seg[D]).mem_phys,
         (current_proc_ptr->seg[S]).mem_len +
         (current_proc_ptr->seg[S]).mem_vir
         - (current_proc_ptr->seg[D]).mem_vir);

/* allocate memory for the segments of the executed image */
proc_clicks = tot_clicks + text_clicks;
if((new_base = alloc_mem(proc_clicks))==NO_MEM)return(EAGAIN);

if(!text_shared){
  (current_proc_ptr->seg[T]).mem_phys = new_base;
  (current_proc_ptr->seg[T]).mem_vir = 0;
  (current_proc_ptr->seg[T]).mem_len = text_clicks;
}
else{
  current_proc_ptr->seg[T] = text_seg;
}

/*  data segment. */
(current_proc_ptr->seg[D]).mem_phys = new_base + text_clicks;
(current_proc_ptr->seg[D]).mem_vir = 0;
(current_proc_ptr->seg[D]).mem_len = data_clicks;

/* stack segment. */
(current_proc_ptr->seg[S]).mem_phys = new_base + text_clicks +
                                      data_clicks + gap_clicks;
(current_proc_ptr->seg[S]).mem_vir = data_clicks + gap_clicks;
(current_proc_ptr->seg[S]).mem_len = stack_clicks;

/* "zero" the bbs, gap, and stack */
bytes = (phys_bytes)(data_clicks + gap_clicks + stack_clicks) << CLICK_SHIFT;
base = (phys_bytes)((new_base+text_clicks) << CLICK_SHIFT);
bss_offset = (data_bytes >> CLICK_SHIFT) << CLICK_SHIFT;
base += bss_offset;
bytes -= bss_offset;
data_vir_addr = (mm_proc_ptr->seg[D]).mem_vir << CLICK_SHIFT;
data_phy_addr = (mm_proc_ptr->seg[D]).mem_phys << CLICK_SHIFT;
```

```
  src = (vir_bytes)zero - data_vir_addr + data_phy_addr;

  while (bytes > 0) {
    count = MIN(bytes, (phys_bytes) sizeof(zero));
    phys_copy(src, base, count);
    base += count;
    bytes -= count;
  }

  /* copy the stacks back from memory server */
  src = dst;
  dst = (vir_bytes)((current_proc_ptr->seg[S]).mem_phys << CLICK_SHIFT);
  dst += (vir_bytes)((current_proc_ptr->seg[S]).mem_len << CLICK_SHIFT);
  dst -= stk_bytes;
  phys_copy(src, dst, stk_bytes);

  current_proc_ptr->proc_flags &= ~SEPARATE;
  current_proc_ptr->proc_flags |= ft;

  /* set the fields of the process descriptor */
  current_proc_ptr->proc_flags &= ~SEPARATE;
  current_proc_ptr->proc_flags |= ft;
  current_proc_ptr->ino = ino;
  current_proc_ptr->dev = dev;
  current_proc_ptr->ctime = ctime;
  current_proc_ptr->sp  = (reg_t)dst;

  /* close the files open originally by the process before
     exec. */
  for(i=0; i<OPEN_MAX; i++){
    if((current_proc_ptr->fp_filp[i])!=NIL_FILP){
      (current_proc_ptr->fp_filp[i])->filp_count--;

      if((current_proc_ptr->fp_filp[i])->filp_count == 0){
        (current_proc_ptr->fp_filp[i])->filp_ino->count--;
      }
      current_proc_ptr->fp_filp[i] = NIL_FILP;
    }
  }
}
```

brk

The function is modified from the function **do_brk** in **MM** in [46].

```
#include "types.h"
```

```
#include "proc.h"

#define DATA_CHANGED 1
#define STACK_CHANGED 2
int brk(vir_bytes addr, pid_t current_pid){
  int i;
  reg_t sp;
  struct proc_t *current_proc_ptr;
  vir_bytes v;
  vir_clicks new_clicks, sp_clicks, gap_base, lower;
  phys_clicks data_phy, stack_phy;
  vir_clicks data_vir, data_len, stack_vir, stack_len;
  long base_of_stack, delta;
  int changed;

  /* find the current process descriptor and its stack
     pointer. */
  for(i=0; i<NR_PROCS; i++){
      if((proc[i].proc_flags & IN_USE != 0)
         && (proc[i].pid == current_pid)){
      current_proc_ptr = &proc[i];
      sp = current_proc_ptr->sp;
      break;
    }
  }

  v = addr;
  new_clicks = (vir_clicks)( ((long)v + CLICK_SIZE - 1) >> CLICK_SHIFT);

  data_vir = (current_proc_ptr->seg[D]).mem_vir;
  data_phy = (current_proc_ptr->seg[D]).mem_phys;
  data_len = (current_proc_ptr->seg[D]).mem_len;

  /* the address will before the start of data segment. */
  if(new_clicks < data_vir)
    return(ENOMEM);

  new_clicks -= data_vir;

  stack_vir = (current_proc_ptr->seg[S]).mem_vir;
  stack_phy = (current_proc_ptr->seg[S]).mem_phys;
  stack_len = (current_proc_ptr->seg[S]).mem_len;
  base_of_stack = (long) stack_vir + (long) stack_len;

  sp_clicks = sp >> CLICK_SHIFT;        /* click containing sp */
  if (sp_clicks >= base_of_stack) return(ENOMEM); /* sp too high */

  /* Compute size of gap between stack and data segments. */
```

```
  delta = (long) stack_vir - (long) sp_clicks;
  lower = (delta > 0 ? sp_clicks : stack_vir);

  /* Add a safety margin for future stack growth. Impossible to do right. */
#define SAFETY_BYTES  (384 * sizeof(char *))
#define SAFETY_CLICKS ((SAFETY_BYTES + CLICK_SIZE - 1) / CLICK_SIZE)
  gap_base = data_vir + new_clicks + SAFETY_CLICKS;
  if (lower < gap_base) return(ENOMEM); /* data and stack collided */

  /* Update data length (but not data orgin) on behalf of brk() system call. */
  if (new_clicks != data_len) {
    data_len = new_clicks;
    changed |= DATA_CHANGED;
  }

  /* Update stack length and origin due to change in stack pointer. */
  if (delta > 0) {
    stack_vir -= delta;
    stack_phy -= delta;
    stack_len += delta;
    changed |= STACK_CHANGED;
  }

  if(changed & STACK_CHANGED){
    (current_proc_ptr->seg[S]).mem_vir = stack_vir;
    (current_proc_ptr->seg[S]).mem_phys = stack_phy;
    (current_proc_ptr->seg[S]).mem_len = stack_len;
  }

  if(changed & DATA_CHANGED) {
    (current_proc_ptr->seg[D]).mem_vir = data_vir;
    (current_proc_ptr->seg[D]).mem_phys = data_phy;
    (current_proc_ptr->seg[D]).mem_len = data_len ;
  }
}
```

A.3.2 Table definitions for kernel data

segment table of main memory segments that hold code, data, or stack of processes:

```
create table segment(
  seg_id  integer NOT NULL,
  virtual_address integer,
  physical_address integer,
  length integer,
  PRIMARY KEY (seg_id)
)
```

proc process descriptor table:

```
create table proc(
  pid  integer NOT NULL,
  parent_pid integer,
  wait_pid integer,
  exit_status integer,
  sig_status integer,
  data_segment integer,
  text_segment integer,
  stack_segment integer,
  ino    integer,
  dev    integer,
  ctime  timestamp,
  stack_pointer integer,
  proc_flags integer,
  PRIMARY KEY (pid),
  FOREIGN KEY (parent_pid) REFERENCES   proc(pid),
  FOREIGN KEY (wait_pid)   REFERENCES   proc(pid),
  FOREIGN KEY (data_segment) REFERENCES segment(seg_id),
  FOREIGN KEY (text_segment) REFERENCES segment(seg_id),
  FOREIGN KEY (stack_segment) REFERENCES segment(seg_id)
)
```

In the table definition, **proc_flags** is an integer whose bits correspond to the flags **SEPARATE**, **WAITING**, and **HANGING**, which are defined together with the process descriptor **proc_t** in Section 4.2.2. Another flag **IN_USE** defined there does not need to be mapped, as it is shown by whether the row "exists" in the table.

inode table of the in-memory inodes of opened files.

```
create table inode (
  inode integer,
  dev   integer,
  ctime timestamp,
  count integer,
  PRIMARY KEY (inode)
)
```

filp file descriptor table:

```
create table filp(
  filp_id  integer NOT NULL,
  inode_id integer,
  position integer,
  mode     integer,
  filp_count integer,
```

```
  PRIMARY KEY (filp_id)
)
```

proc_filp table that associates processes to their file descriptors:

```
create table proc_filp(
  proc_id  integer,
  flip_id  integer,
  FOREIGN KEY (proc_id) REFERENCES proc(pid),
  FOREIGN KEY (proc_id) REFERENCES filp(filp_id)
)
```

hole free memory hole table:

```
create table hole (
  base integer,
  len  integer
)
```

pid table of integers from 1 to 30000 to get a process identifier:

```
create table pid (
  pid  integer
)
```

ready_user_proc table of user processes that are ready to run:

```
create table ready_user_proc (
  pid integer,
  FOREIGN KEY (pid) REFERENCES proc(pid)
)
```

A.3.3 C/SQL programs for the system calls

fork

```
int fork(int pid){
  EXEC SQL BEGIN DECLARE SECTION;
    int proc_in_use;

    int parent_pid;
    int child_pid;
    int wait_pid;

    int available_holes;
    int hole_base;
    int hole_len;
```

```
  int mem_len_stack;
  int mem_len_data;
  int mem_len_text;

  int mem_vir_stack;
  int mem_vir_data;
  int mem_vir_text;

  int mem_phys_data;
  int mem_phys_stack;
  int mem_phys_text;

  int prog_clicks;

  int seg_id_stack;
  int seg_id_data;
  int seg_id_text;

  int curr_proc_flags;

  int exit_status;
  int sig_status;

  int ino;
  int dev;
  char change_time[27];
  int filp_id;

  int stack_pointer;
EXEC SQL END DECLARE SECTION;

int i;
int child_base;
int child_abs;
int parent_abs;
int ret_value;

/* wether we reach the upperlimit of number of processes? */
EXEC SQL
SELECT count(*) into :proc_in_use
FROM proc;

if (proc_in_use >= NR_PROCS) {
  ret_value = EAGAIN;
  goto out;
}

/* select all fields from the proc */
```

```
parent_id = pid;
EXEC SQL
SELECT data_segment into :seg_id_data,
       stack_segment into :seg_id_stack,
       text_segment into :seg_id_stack,
       ino    into :ino,
       dev    into :dev,
       change_time into :change_time,
       proc_flags into :curr_proc_flags,
       wait_pid into :wait_pid,
       stack_pointer into :stack_pointer
FROM proc
WHERE proc.pid = :parent_id;

/* calculate the memory that is needed for the new
   process */
/* stack */
EXEC SQL
SELECT segment.length into :mem_len_stack,
       segment.virtual_address into :mem_vir_stack
FROM  segment
AND   segment.seg_id = :seg_id_stack;

/* data */
EXEC SQL
SELECT segment.virtual_address into :mem_vir_data,
       segment.physical_address into :mem_phys_data
FROM segment
WHERE  segment.seg_id = :seg_id_data;

/* calculate the full memory */
/* for text, it is either shared (in same physical
   memory) by multiple processes, or merged with
   data segment, so we handle it later. */
mem_len_data = mem_vir_stack - mem_vir_data
proc_click = mem_len_stack + mem_len_data;

/* get memory chunk from the holes and adjust the hole size */
EXEC SQL DECLARE hole_cursor CURSOR FOR
  SELECT len, base
  FROM hole
  WHERE len >= :proc_click
FOR UPDATE;

EXEC SQL OPEN hole_cursor;

EXEC SQL FETCH hole_cursor into :hole_len, :hole_base;
```

```
if(SQL_CODE != 0){
  EXEC CLOSE hole_cursor;
  ret_value = EGAGAIN;
  goto err_label;
}

child_base = hole_base;
hole_len -= proc_click;
hole_base += proc_click;

if(hole_len>0){
  EXEC SQL
  UPDATE hole SET len = :hole_len, base = :hole_base
  WHERE CURRENT OF CURSOR hole_cursor;
}
else{
  /* if the segment takes the entire hole, the hole is
   deleted */
  EXEC SQL
  DELETE FROM hole
  WHERE CURRENT OF CURSOR hole_cursor;
}

EXEC SQL CLOSE hole_cursor;

/* copy the content of main memory */
child_abs = child_base << CLICK_SHIFT;
parent_abs = mem_phys_data << CLICK_SHIFT;
phys_copy(parent_abs, child_abs, prog_bytes);

/* now get a child id */
/* it must be different from any existing process */
EXEC SQL DECLARE pid_cursor FOR
SELECT pid
FROM pids
WHERE NOT EXISTS
  (SELECT *
   FROM proc
   WHERE proc.pid = pids.pid
  );

EXEC OPEN pid_cursor;
EXEC fetch pid_cursor into :child_pid;
EXEC close pid_cursor;

/* set the address of the child's segments */
mem_phys_stack = childbase + mem_len_data;
mem_phys_data = childbase;
```

```
/* get segment ids */
/* times 10 to get room for text segment id */
/* overwrite the value of the variables, which are the parent's segments,
   except the text segment if it shared. */
seg_id_data = mem_phys_data;
seg_id_stack = mem_phys_stack;
seg_id_data *= 10;
seg_id_stack*= 10 + 2;

/* insert into segment table */
EXEC SQL
INSERT INTO segment
VALUES(:seg_id_stack, :mem_vir_stack, :mem_phys_stack, :mem_len_stack);

EXEC SQL
INSERT INTO segment
VALUES(:seg_id_data, :mem_vir_data, :mem_phys_data, :mem_len_data);

/* if non-separate text and data */
if(!(curr_proc_flags & SEPARATE)){
  /* make the segment id */
  seg_id_text =  seg_id_data + 1;

  /* text segment and data segment are not separated */
  EXEC SQL
  INSERT INTO segment
  VALUES(:seg_id_text, :mem_vir_text, :mem_phys_data, :mem_len_text);
}
/* else share the original text segment */

/* insert a row into the process table */
EXEC SQL
INSERT INTO
  proc(pid,
       parent_id,
       wait_pid,
       exit_status,
       sig_status,
       data_segment,
       text_segment,
       stack_segment,
       ino,
       dev,
       change_time,
       stack_pointer,
       proc_flags)
VALUES(:child_pid, :parent_pid, :wait_pid,
```

```
        :exit_status, :sig_status,
        :seg_id_data, :seg_id_text, :seg_id_stack,
        :ino, :dev, :change_time,
        :stack_pointer, :curr_proc_flags);

  /* share the files with the child */
  EXEC SQL DECLARE proc_filp_cursor CURSOR FOR
    SELECT filp_id
    FROM proc_filp
    WHERE proc_id = :parent_pid;

  EXEC SQL OPEN proc_filp_cursor;

  if(SQLCODE == 0){
    while(true){
     EXEC SQL FETCH proc_filp_cursor into :filp_id;
     if(SQLCODE != 0 ) break;

     EXEC SQL
     INSERT INTO proc_filp
     VALUES(:child_pid, :filp_id);

     EXEC SQL
     UPDATE filp SET filp_count = filp_count + 1
     WHERE filp_id = :filp_id;
    }
  }

  EXEC SQL CLOSE proc_filp_cursor;

  /* add to ready queue */
  EXEC SQL
    INSERT INTO ready_user_proc
    VALUES (:child_pid);

out:
  return (ret_value);
}
```

exit

```
exit(int pid) {
  EXEC SQL BEGIN DECLARE SECTION;
    int current_pid;
    int parent_pid;
    int wait_pid;
```

```
int exit_status;
int sig_status;

int data_segment;
int text_segment;
int stack_segment;

int proc_flags;

int file_count;
int text_share_count;
int generic_count;

int mem_vir_text;
int mem_phy_text;
int mem_len_text;

int mem_vir_data;
int mem_phy_data;
int mem_len_data;

int mem_vir_stack;
int mem_phy_stack;
int mem_len_stack;

int mem_phy_base;
int mem_phy_len;
int mem_phy_end;

int parent_wait_pid;
int parent_proc_flags;

int init_proc_pid;
int child_id;
int child_proc_flags;

int ino;
int dev;
int ctime;

int base1;
int len1;

int base2;
int len2;

int temp_pid;
int temp_proc_flags;
```

```
  int temp_ino;
  int temp_dev;
  int temp_c_time;
EXEC SQL BEGIN DECLARE SECTION;

int flag;
int right_child;
int parent_waiting;
boolean delete_text;
int temp_len;

current_pid = pid;

/* first close all open files */
/* we simplify file close to delete */
/* decrement the reference count in filp */
EXEC SQL UPDATE filp set filp_count = filp_count - 1
WHERE EXISTS (
  SELECT * FROM proc_filp
  WHERE proc_filp.proc_id = :current_pid
  AND   proc_filp.filp_id = filp.filp_id
);

/* decrement the count of the inode in memory of an
   open file */
EXEC SQL UPDATE inode set count = count - 1
WHERE EXISTS (
  SELECT * from filp
  WHERE filp.filp_count = 0
  AND filp.inode_id = inode.inode
)

/* delete from the proc-filp relationship */
EXEC SQL DELETE FROM proc_filp
WHERE proc_id = :current_pid;

/* delete the entries which has no
   process that references it */
EXEC SQL DELETE FROM filp
WHERE filp_count = 0;

/* delete the entries of inodes which has
   been closed by all processes */
EXEC  SQL DELETE FROM inode
WHERE count = 0;

/* get all info of the exiting process */
```

```
EXEC SQL SELECT
  parent_pid into :parent_pid,
  wait_pid into :wait_pid,
  exit_status into :exit_status,
  sig_status into :sig_status,
  data_segment into :data_segment,
  text_segment into :text_segment,
  stack_segment into :stack_segment,
  proc_flags into :proc_flags,
  ino into :ino,
  dev into :dev,
  ctime into :ctime
FROM proc
WHERE pid = :current_pid;

/* get the process out of the ready queue */
DELETE FROM ready_user_proc
WHERE pid = :current_pid

/* delete segments */
delete_text = FALSE;

EXEC SQL DECLARE proc_cursor CURSOR FOR
  SELECT pid, ino, dev, ctime, proc_flags
  FROM proc;

EXEC OPEN proc_cursor;

while(SQLCODE == 0){
  EXEC FETCH proc_cursor
  into :temp_pid, :temp_ino, :temp_dev, :temp_ctime,
       :temp_proc_flags;

  if( (temp_proc_flags & (SEPARATE | HANGING)) !=
       SEPARATE ) continue;
  if((temp_pid == pid) || (temp_ino != ino) ||
     (temp_dev != dev) || (temp_ctime != ctime))
    continue;
  else{
    delete_text = TRUE;
    break;
  }
}
EXEC CLOSE proc_cursor;

/* check the case of shared text segment */
if(delete_text){
  /* get the physical address and length of the text */
```

```
EXEC SQL SELECT physical_address into :mem_phy_text,
                length into :mem_len_text
FROM segment
WHERE segment.sid = :text_segment;

/* next, we free the text_segment and insert the
   physical segment into the hole table */

/* fist see whether we can merge it to other holes. */
flag = FALSE;

/* now we have the end of the text segment */
mem_phy_end = mem_phy_text + mem_len_text;

/* can we merge it with the hole after it */
EXEC SQL DECLARE hole_cursor1 CURSOR FOR
  SELECT base, len
  FROM hole
  WHERE base = :mem_phy_end
FOR UPDATE;

EXEC SQL OPEN hole_cursor1;
EXEC SQL FETCH hole_cursor1 INTO :base1, :len1;

if(SQLCODE == 0){
  /* we can merge */
  flag = TRUE;
  len1 = len1 + mem_len_text;
  base1 = mem_phy_text;

  /* merge */
  EXEC SQL UPDATE hole
  SET base = :base1,
      len  = :len1
  WHERE current of hole_cursor1;
}

/* can we merge it with the hole ahead of it */
EXEC SQL DECLARE hole_cursor_2 FOR
  SELECT base, len
  FROM hole
  WHERE base + len = :mem_phy_text
FOR UPDATE;

EXEC SQL OPEN hole_cursor_2;
EXEC SQL FETCH hole_cursor_2 to :base2, :len2;

/* we can merge */
```

```
  if(SQLCODE == 0){
    /* Have we already merged it with the one afterwards. */
    if(!flag){
      /* no. so merge it to the one in front of it. */
      flag = TRUE;

      EXEC SQL UPDATE hole
      SET len = :len2 + :mem_len_text
      WHERE current of hole_cursor_2;
    }
    else{
      /* the hole afterwards is from the merging in
         the above step. */
      /* merge the hole with the hole after it. */
      EXEC SQL UPDATE hole
      SET len = :len2 + :len1
      WHERE current of hole_cursor_2;

      EXEC SQL DELETE FROM hole
      WHERE current of hole_cursor_1;
    }
  }

  EXEC SQL CLOSE hole_cursor_1;
  EXEC SQL CLOSE hole_cursor_2;

  if(!flag){
    /* no merge at all, add the segment to the hole table. */
    EXEC SQL INSERT INTO hole
    VALUES(:mem_phy_text, :mem_len_text);
  }
}

/* handle the stack and data */
/* they are contiguous physical segments */
/* so we return them together */
EXEC SQL SELECT mem_phy_data into :mem_phy_data,
                mem_vir_data into :mem_vir_data
FROM proc, segment
WHERE segment.seg_id = :data_segment
AND proc.pid = :current_pid;

EXEC SQL SELECT mem_vir_stack into :mem_vir_stack,
                mem_len_stack into :mem_len_stack
FROM segment
WHERE segment.seg_id = :stack_segment

temp_len = mem_vir_stack -
```

```
          mem_vir_data + mem_len_stack;
mem_phy_end = mem_phy_data + temp_len;

flag = FALSE;

/* merge with the one after it */
EXEC SQL DECLARE CURSOR hole_cursor_3 FOR
  SELECT base, len
  FROM hole
  WHERE base = :mem_phy_end
FOR UPDATE;

EXEC SQL OPEN hole_cursor_3;
EXEC SQL FETCH hole_cursor_3 INTO :base1, :len1;

if(SQLCODE = 0){
  flag = TRUE;
  base1 = mem_phy_data;
  len1 = len1 + temp_len;

  EXEC SQL UPDATE hole
  SET base = :base1,
      len  = :len1
  WHERE current of hole_cursor_3;
}

/* merge the one ahead of it */
EXEC SQL DECLARE CURSOR hole_cursor_4 FOR
SELECT base, len
FROM hole
WHERE base + len = :mem_phy_data;
FOR UPDATE;

EXEC SQL OPEN hole_cursor_4;
EXEC SQL OPEN hole_cursor_4 into :base2, :len2;

/* has the segment already merged with the one
   behind it? */
if(SQLCODE == 0){
  if(!flag){
    flag = TRUE;
    len2 += temp_len;

    EXEC SQL UPDATE hole
    SET len = :len2
    WHERE current of hole_cursor_4;
  }
  else{
```

```
    /* it may cause merge of three segments */

    EXEC SQL UPDATE hole
    SET len = :len1 + :len2
    WHERE current of hole_cursor_4;

    EXEC SQL DELETE FROM hole
    WHERE current of hole_cursor_3;
  }
}

EXEC SQL CLOSE hole_cursor_3;
EXEC SQL CLOSE hole_cursor_4;

if(!flag){
  /* no merge at all, so insert the hole
     into the table. */
  mem_phy_len = temp_len;
  EXEC SQL INSERT INTO hole
  VALUES(:mem_phy_data, :mem_phy_len);
}

/* set the segments pointers to NULL */
/* this is because they are foreign keys */
/* then we will delete the three segments in
   the segment table */
EXEC SQL UPDATE proc
         SET stack_segment = NULL,
             text_segment = NULL,
             data_segment = NULL;

/* delete the segments in the segment table */
EXEC SQL DELETE FROM segment
WHERE seg_id = :stack_segment
   OR seg_id = :data_segment;

if(delete_text){
  EXEC SQL DELETE FROM segment
  WHERE seg_id = :text_segment;
}

/* Let INIT_PROC adopt my children */
EXEC SQL DECLARE CURSOR init_cursor FOR
SELECT proc_flags
FROM proc
WHERE pid = :init_proc_pid
FOR UPDATE;
```

```
EXEC SQL OPEN init_cursor;
EXEC FETCH init_cursor into :parent_proc_flags;

parent_waiting = parent_proc_flags & WAITING;

/* if parent is waiting */
if(parent_waiting){
  EXEC SQL DECLARE child_proc_cursor CURSOR FOR
    SELECT proc_flags FROM proc
    WHERE  parent_pid = :current_pid
  FOR UPDATE;

  EXEC SQL OPEN child_proc_cursor;

  while(SQLCODE == 0){
    EXEC SQL FETCH child_proc_cursor into :child_proc_flags;

    if(child_proc_flags & HANGING){
      /* delete the child if the parent is waiting and the child
         has already exited */
      DELETE FROM proc
      WHERE CURRENT OF child_proc_cursor;

      parent_proc_flags &= ~WAITING;

      /* clear the waiting flag */
      EXEC SQL UPDATE proc
           SET proc_flags = :parent_proc_flags;
      WHERE current of init_cursor;

      break;
    }
  }

  EXEC SQL CLOSE init_cursor;
  EXEC SQL CLOSE child_proc_cursor;
}

EXEC SQL UPDATE proc
SET parent_pid = :init_proc_id
WHERE parent_pid = :current_pid;

/* Is my parent waiting? */
EXEC SQL DECLARE CURSOR parent_cursor FOR
SELECT wait_pid into :parent_wait_pid,
               proc_flags   into :parent_proc_flags
FROM proc
WHERE proc.pid = :parent_pid;
```

```
  FOR UPDATE;

  EXEC SQL OPEN parent_cursor;
  EXEC SQL FETCH parent_cursor INTO :parent_wait_pid,
                                    :parent_proc_flags;

  parent_waiting = parent_proc_flags & WAITING;

  /* wait for any child or wait for me? */
  if(parent_wait_pid == -1 || parent_wait_pid == current_pid){
    right_child = TRUE;
  }
  else{
    right_child = FALSE;
  }

  if(parent_waiting && right_child){
    /* parent is waiting for me */
    /* delete the table record for the process */
    DELETE FROM proc
    WHERE pid = :current_pid;

    parent_proc_flags &= ~WAITING;

    /* clear the waiting flag */
    EXEC SQL UPDATE proc
         SET proc_flags =: parent_proc_flags
    WHERE CURRENT OF parent_cursor;
  }
  else{
    /* parent not waiting, set the HANGING flag */
    proc_flags |= HANGING;

    EXEC SQL UPDATE proc
         SET proc_flags = :proc_flags
    WHERE CURRENT OF parent_cursor;
    /* the entry will be deleted when the parent calls wait */
  }
  EXEC CLOSE parent_cursor;
}
```

waitpid

```
int waitpid(int, pid, int pidarg)
/* pidarg == -1, wait for any child,
   pidarg > 0,   wait for a special pid */
{
```

```
EXEC SQL BEGIN DECLARE SECTION;
  int current_pid;
  int wait_pid;

  int child_proc_flags;
  int parent_proc_flags;
  int wait_pid;

  int child_count;
EXEC SQL END DECLARE SECTION;

if(pidarg != -1 && pidarg <= 0) return (ERROR);

wait_pid = pidarg;

/* get the parent. */
current = pid;
EXEC SQL DECLARE CURSOR parent_flag_cursor FOR
SELECT proc_flags
FROM proc
WHERE pid = :current_pid
FOR UPDATE;

EXEC OPEN parent_flag_cursor;
EXEC FETCH parent_flag_cursor into :parent_proc_flags;

if(pidarg == -1){
  /* wait for any child */
  /* get a children */
  EXEC SQL DECLARE child_proc_cursor FOR
    SELECT proc_flags
    FROM proc
    WHERE parent_pid = :current_pid
  FOR UPDATE;

  EXEC SQL OPEN child_proc_cursor;

  if(SQLCODE != 0) {
    /* Error in getting children */
    return(ECHILD);
  }

  while(SQLCODE == 0){
    EXEC SQL FETCH child_proc_cursor into :child_proc_flags;

    if(child_proc_flags & HANGING){
      /* delete the child if the parent is waiting and the child
         has already exited */
```

```
      DELETE FROM proc
      WHERE CURRENT OF child_proc_cursor;

      parent_proc_flags &= ~WAITING;

      /* clear the waiting flag. */
      EXEC SQL UPDATE proc
      SET proc_flags := parent_proc_flags
      WHERE CURRENT OF parent_flag_cursor;

      EXEC SQL CLOSE child_proc_cursor;
      EXEC SQL CLOSE parent_flag_cursor;
      return(OK);
    }
  }
}
else{  /* pidarg is wait_pid */
  /* wait for a certain pid */
  /* get the children */
  EXEC SQL DECLARE child_proc_cursor FOR
    SELECT proc_flags
    FROM proc
    WHERE parent_pid = :current_pid
    AND  pid = :wait_pid;
  FOR UPDATE;

  EXEC SQL OPEN child_proc_cursor;

  if(SQLCODE != 0){
    /* no qualified children, must be an error */
    return(ECHILD);
  }

  EXEC SQL FETCH child_proc_cursor into :child_proc_flags;

  if(child_proc_flags & HANGING){
    /* delete the child if the parent is waiting and the child
       has already exited */
    DELETE FROM proc
    WHERE CURRENT OF child_proc_cursor;

    parent_proc_flags &= ~WAITING;

    /* clear the waiting flag. */
    EXEC SQL UPDATE proc
    SET proc_flags := parent_proc_flags
    WHERE CURRENT OF parent_flag_cursor;
```

```
      EXEC SQL CLOSE child_proc_cursor;
      EXEC SQL CLOSE parent_flag_cursor;
      return(OK);
    }
  }

  /* we have qualified children, but can not find a child that has
     exited */
  /* set the parent's status to waiting */
  parent_proc_flags |= WAITING;

  EXEC SQL UPDATE proc
  SET proc_flags := parent_proc_flags
  WHERE CURRENT OF parent_flag_cursor;

  EXEC SQL CLOSE child_proc_cursor;
  EXEC SQL CLOSE parent_flag_cursor;

  return (OK);
}
```

exec

```
#define MM_PROC_NR 0
#define ARG_MAX  4096

into exec(int pid,
          int stack_ptr,
          int stk_bytes,
          int data_bytes,
          int bss_bytes,
          int text_bytes,
          int tot_bytes,
          int ino_param,
          int dev_param,
          int ctime_param,
          int ft) {
  EXEC SQL BEGIN DECLARE SECTION;
    int current_pid;

    int mm_proc_pid;

    int stack_segment_phy;
    int stack_segment_vir;
    int stack_segment_len;

    int data_segment_phy;
```

```
int data_segment_vir;
int data_segment_len;

int text_segment_phy;
int text_segment_vir;
int text_segment_len;

int segment_end;

int text_count;

int text_seg_id;
int data_seg_id;
int stack_seg_id;

int old_text_seg_id;
int old_data_seg_id;
int old_stack_seg_id;

int proc_flags;

int max_hole_size;

int hole_base;
int hole_len;

int ino;
int dev;
int ctime;

int old_ino;
int old_dev;
int old_ctime;

int temp_pid;
int temp_ino;
int temp_dev;
int temp_ctime;

int proc_clicks;
int count_generic;

int stack_pointer;

int base1;
int len1;

int base2;
```

```
  int len2;
EXEC SQL END DECLARE SECTION;

char mbuf[ARG_MAX];
int  src, dst;
boolean text_shared;
boolean delete_old_text;
boolean flag;
int new_base;
int text_clicks, data_clicks, stack_clicks,
    total_clicks, gap_clicks;
int base, bytes, count, bss_offset;
static char zero[1024];
int mm_data_phy, mm_data_vir;
int temp_len;
int ft;

ino = ino_param;
dev = dev_param;
ctime = ctime_param;

/* copy the content of the stack to the mem space of mm task */
current_pid = pid;
EXEC SQL SELECT
  text_segment into :old_text_seg_id,
  stack_segment into :old_stack_seg_id,
  data_segment into :old_data_seg_id,
  proc_flags  into :proc_flags,
  ino     into :old_ino,
  dev     into :old_dev,
  ctime   into :old_ctime
FROM proc
WHERE proc_id = :current_pid;

EXEC SQL SELECT segment.physical_address into :data_segment_phy,
                segment.virtual_address into :data_segment_vir
FROM segment
WHERE seg_id = old_data_seg_id;

mm_data_vir = data_segment_vir << CLICK_SHIFT;
mm_data_phy = data_segment_phy << CLICK_SHIFT;
src = stack_ptr - mm_data_vir + mm_data_phy;

mm_proc_pid = MM_PROC_NR;
EXEC SQL SELECT segment.physical_address into :data_segment_phy,
                segment.virtual_address into :data_segment_vir
FROM proc, segment
WHERE segment.seg_id = proc.data_segment
```

```
AND proc_id = :mm_proc_pid;

mm_data_vir = data_segment_vir << CLICK_SHIFT;
mm_data_phy = data_segment_phy << CLICK_SHIFT;
dst = mbuf - mm_data_vir + mm_data_phy;

phys_copy(src, dst, stack_bytes);

/* find text share  */
text_shared = FALSE;

EXEC SQL DECLARE proc_cursor CURSOR FOR
SELECT pid, ino, dev, ctime, proc_flags, text_segment
FROM proc;

EXEC OPEN proc_cursor;

while(SQLCODE == 0){
  EXEC FETCH proc_cursor
  into :temp_pid, :temp_ino, :temp_dev, :temp_ctime,
       :temp_proc_flags, :text_seg_id;

  if( (temp_proc_flags & (SEPARATE | HANGING)) !=
       SEPARATE ) continue;
  if((temp_pid == current_pid) || (temp_ino != ino) ||
     (temp_dev != dev) || (temp_ctime != ctime))
    continue;
  else{
    text_shared = TRUE;
    text_bytes = 0;
    break;
  }
}
EXEC CLOSE proc_cursor;

/* new segments for the executed proc */
text_clicks = ((unsigned long) text_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
data_clicks = (data_bytes + bss_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
stack_clicks = (stk_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
tot_clicks = (tot_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
gap_clicks = tot_clicks - data_clicks - stack_clicks;
if ( (int) gap_clicks < 0) return(ENOMEM);

/* whether there is a hole that can fit the memory */
EXEC SQL SELECT max(len) into :max_hole_size
FROM hole;

if(max_hole_size < tot_clicks + text_clicks) return (EAGAIN);
```

```
/* first release the old segment */
/* text segment */
delete_old_text = TRUE;

EXEC SQL DECLARE proc_cursor CURSOR FOR
  SELECT pid, ino, dev, ctime, proc_flags
  FROM proc;

EXEC OPEN proc_cursor;

while(SQLCODE == 0){
  EXEC FETCH proc_cursor
  into :temp_pid, :temp_ino, :temp_dev, :temp_ctime,
       :temp_proc_flags;

  if( (temp_proc_flags & (SEPARATE | HANGING)) !=
       SEPARATE ) continue;
  if((temp_pid == current_pid) || (temp_ino != old_ino) ||
     (temp_dev != old_dev) || (temp_ctime != old_ctime))
    continue;
  else{
    delete_old_text = FALSE;  /* it is shared, cannot delete it */
    break;
  }
}
EXEC CLOSE proc_cursor;

/* check the case of shared text segment. */
if(delete_old_text){
  /* get the physical address and length of the text */
  EXEC SQL SELECT physical_address into :text_segment_phy,
                  length into :text_segment_len
  FROM segment
  WHERE segment.sid = :old_text_seg_id;

  /* next, we free the text_segment and insert the
     physical segment into the hole table. */
  /* fist see whether we can merge it to other holes. */
  flag = FALSE;

  /* now we have the end of the text segment. */
  segment_end = text_segment_phy + text_segment_len;
  /* can we merge it with the hole after it? */
  EXEC SQL DECLARE hole_cursor1 CURSOR FOR
    SELECT base, len
    FROM hole
    WHERE base = :segment_end
```

```
FOR UPDATE;

EXEC SQL OPEN hole_cursor1;

EXEC SQL FETCH hole_cursor1 INTO :base1, :len1;

if(SQLCODE == 0){
  /* we can merge */
  flag = TRUE;

  len1 = len1 + text_segment_len;
  base1 = text_segment_phy;

  /* merge */
  EXEC SQL UPDATE hole
  SET base = :base1,
      len  = :len1
  WHERE current of hole_cursor1;
}

/* can we merge it with the hole ahead of it? */
EXEC SQL DECLARE hole_cursor_2 FOR
  SELECT base, len
  FROM hole
  WHERE base + len = :text_segment_phy
FOR UPDATE;

EXEC SQL OPEN hole_cursor_2;
EXEC SQL FETCH hole_cursor_2 to :base2, :len2;

/* we can merge */
if(SQLCODE == 0){
  /* Have we already merged it with the one afterwards. */
  if(!flag){
    /* no. so merge it to the one in front of it. */
    flag = TRUE;
    EXEC SQL UPDATE hole
    SET len = :len2 + :text_segment_len
    WHERE current of hole_cursor_2;
  }
  else{
    /* the hole afterwards is from the merging in
       the above step. */
    /* merge the hole with the hole after it. */
    EXEC SQL UPDATE hole
    SET len = :len2 + :len1
    WHERE current of hole_cursor_2;
```

```
      EXEC SQL DELETE FROM hole
      WHERE current of hole_cursor_1;
    }
  }

  EXEC SQL CLOSE hole_cursor_1;
  EXEC SQL CLOSE hole_cursor_2;

  if(!flag){
    /* no merge at all, add the segment to the hole table. */
    EXEC SQL INSERT INTO hole
    VALUES(:text_segment_phy, :text_segment_len);
  }
}

/* handle the stack and data */
/* they are contiguous physical segments */
/* so we return them together */
EXEC SQL SELECT mem_phy_data into :data_segment_phy,
                mem_vir_data into :data_segment_vir
FROM segment
WHERE segment.seg_id = :old_data_seg_id;

EXEC SQL SELECT mem_vir_stack into :stack_segment_vir,
                mem_len_stack into :stack_segment_len
FROM segment
WHERE segment.seg_id = :old_stack_seg_id;

temp_len = stack_segment_vir - data_segment_vir
           + stack_segment_len;
segment_end = data_segment_phy + temp_len;

flag = FALSE;

/* merge with the one after it */
EXEC SQL DECLARE CURSOR hole_cursor_3 FOR
  SELECT base, len
  FROM hole
  WHERE base = :segment_end
FOR UPDATE;

EXEC SQL OPEN hole_cursor_3;

EXEC SQL FETCH hole_cursor_3 INTO :base1, :len1;

if(SQLCODE = 0){
  flag = TRUE;
  base1 = data_segment_phy;
```

```
  len1 = len1 + temp_len;

  EXEC SQL UPDATE hole
  SET base = :base1,
      len  = :len1
  WHERE current of hole_cursor_3;
}

/* merge the one ahead of it */
EXEC SQL DECLARE CURSOR hole_cursor_4 FOR
SELECT base2, len2
FROM hole
WHERE base + len = :segment_data_phy;
FOR UPDATE;

EXEC SQL OPEN hole_cursor_4;
EXEC SQL OPEN hole_cursor_4 into :base2, :len2;

/* has the segment already merged with the one
   behind it? */
if(SQLCODE == 0){
  if(!flag){
    flag = TRUE;
    len2 += temp_len;

    EXEC SQL UPDATE hole
    SET len = :len2
    WHERE current of hole_cursor_4;
  }
  else{
    /* it may cause merge of three segments */
    EXEC SQL UPDATE hole
    SET len = :len1 + :len2
    WHERE current of hole_cursor_4;

    EXEC SQL DELETE FROM hole
    WHERE current of hole_cursor_3;
  }
}

EXEC SQL CLOSE hole_cursor_3;
EXEC SQL CLOSE hole_cursor_4;

if(!flag){
  len1 = temp_len;

  EXEC SQL INSERT INTO hole
  VALUES(:mem_phy_data, :len1);
```

```
}

/* set the segments pointers to NULL */
/* this is because they are foreigh keys */
/* then we will delete the three segments in
   the segment table */
EXEC SQL UPDATE proc
         SET stack_segment = NULL,
             text_segment = NULL,
             data_segment = NULL
WHERE proc_id = :current_pid;

/* delete the segments in the segment table */
EXEC SQL DELETE FROM segment
WHERE seg_id = :old_stack_seg_id
   OR seg_id = :old_data_seg_id;

if(delete_old_text){
  EXEC SQL DELETE FROM segment
  WHERE seg_id = :old_text_seg_id;
}

proc_clicks = tot_clicks + text_clicks;

/* get memory chunk from the holes and adjust the hole size */
EXEC SQL DECLARE hole_cursor CURSOR FOR
  SELECT len, base
  FROM hole
  WHERE len >= :proc_clicks
FOR UPDATE;

EXEC SQL OPEN hole_cursor;
EXEC SQL FETCH hole_cursor into :hole_len, :hole_base;

new_base = hole_base;
hole_len -= proc_click;
hole_base += proc_click;

if(hole_len > 0){
  EXEC SQL
  UPDATE hole SET len = :hole_len, base = :hole_base
  WHERE CURRENT OF CURSOR hole_cursor;
}
else{
  /* if the segment takes the entire hole, the hole is
   deleted */
  EXEC SQL
  DELETE FROM hole
```

```
  WHERE CURRENT OF CURSOR hole_cursor;
}

EXEC SQL CLOSE hole_cursor;

/* if the text is not shared yet, insert a segment for it. */
if(!text_shared){
  text_segment_phy = new_base;
  text_segment_vir = 0;
  text_segment_length = text_clicks;
  text_seg_id = new_base * 10 + 1;

  EXEC SQL INSERT INTO segment
  (seg_id, virtual_address, physical_address, length)
  VALUES(:text_seg_id, :text_segment_vir, :text_segment_phy,
         :text_segment_length);
}

data_segment_phy = new_base + text_clicks;
data_segment_vir = 0;
data_segment_length = data_clicks;
data_seg_id = data_segment_phy * 10;

EXEC SQL INSERT INTO segment
(seg_id, virtual_address, physical_address, length)
VALUES(:data_seg_id, :data_segment_vir, :data_segment_phy,
       :data_segment_length);

stack_segment_phy = new_base + text_clicks + data_clicks + gap_clicks;
stack_segment_vir = data_clicks + gap_clicks;
stack_segment_length = stack_clicks;
stack_seg_id = stack_segment_phy * 10 + 2;

EXEC SQL INSERT INTO segment
(seg_id, virtual_address, physical_address, length)
VALUES(:stack_seg_id, :stack_segment_vir, :stack_segment_phy,
       :stack_segment_length);

/* zero the segments in memory */
bytes = (phys_bytes)(data_clicks + gap_clicks + stack_clicks) << CLICK_SHIFT;
base = (phys_bytes)data_segment_phy << CLICK_SHIFT;
bss_offset = (data_bytes >> CLICK_SHIFT) << CLICK_SHIFT;
base += bss_offset;
bytes -= bss_offset;

while (bytes > 0) {
  count = MIN(bytes, (phys_bytes) sizeof(zero));
  sys_copy(ABS, 0, zero - mm_data_vir + mm_data_phy, ABS, 0, base, count);
```

```
  base += count;
  bytes -= count;
}

/* recovery the stacks from the MM server. */
src = dst;
dst = stack_segment_phy << CLICK_SHIFT;
dst += stack_segment_len << CLICK_SHIFT;
dst -= stk_bytes;
stack_pointer = dst;

phys_copy(src, dst, stk_bytes);

proc_flags &= ~SEPARATE;
proc_flags |= ft;

/* update the process descriptor. */
EXEC SQL UPDATE proc
SET text_segment = :text_seg_id,
    data_segment = :data_seg_id,
    stack_segment = :stack_seg_id
    ino = :ino,
    ctime = :ctime,
    dev = :dev,
    stack_pointer := stack_pointer;
    proc_flags = :proc_flags,
WHERE proc.pid = :current_pid;

/* close open files of the process before calling exec */
EXEC SQL UPDATE filp set filp_count = filp_count - 1
WHERE EXISTS (
  SELECT * FROM proc_filp
  WHERE proc_filp.proc_id = :current_pid
  AND   proc_filp.filp_id = filp.filp_id
);

/* decrement the count of the inode in memory of an
   open file */
EXEC SQL UPDATE inode set count = count - 1
WHERE EXISTS (
  SELECT * from filp
  WHERE filp.filp_count = 0
  AND filp.inode_id = inode.inode
)

/* delete from the proc-filp relationship */
EXEC SQL DELETE FROM proc_filp
WHERE proc_id = :current_pid;
```

```
  /* The above update filp may changes
     filp_count to 0
  */
  /* delete the entries which has no
     process that references it */
  EXEC SQL DELETE FROM filp
  WHERE filp_count = 0;

  / * delete the inode if no process is
      referencing to it */
  EXEC SQL DELETE FROM inode
  WHERE count = 0;
}
```

brk

```
#define DATA_CHANGED 1
#define STACK_CHANGED 2

brk(int addr, int pid){
  EXEC SQL BEGIN DECLARE SECTION;
    int current_pid;

    int stack_pointer;

    int data_phy;
    int data_vir;
    int data_len;
    int data_seg;

    int stack_phy;
    int stack_vir;
    int stack_len;

    int stack_seg;
  EXEC SQL BEGIN DECLARE SECTION;

  vir_bytes v, new_sp;
  vir_clicks new_clicks, sp_click, gap_base, lower;

  long base_of_stack, delta;
  int changed;

  /* get the segment info of the process. */
  current_pid = pid;
  EXEC SQL SELECT
```

```
    data_segment into :data_seg,
    stack_segment into :stack_seg,
    stack_pointer into :stack_pointer
  FROM proc
  WHERE proc.pid = :current_pid;

  v = addr;
  new_clicks = (vir_clicks) ( ((long)v + CLICK_SIZE - 1) >> CLICK_SHIFT);

  EXEC SQL SELECT
    virtual_address into :data_vir,
    physical_address into :data_phy,
    length into :data_len
  FROM segment
  WHERE proc.data_segment = :data_seg;

  if(new_clicks < data_vir)
    return(ENOMEM);

  new_clicks -= data_vir;

  EXEC SQL SELECT
    virtual_address into :stack_vir,
    physical_address into :stack_phy,
    length into stack_len
  FROM segment
  WHERE proc.stack_segment = :stack_seg;

  base_of_stack = (long) stack_vir + (long) stack_len;
  sp_clicks = stack_pointer >> CLICK_SHIFT; /* click containing sp */
  if (sp_clicks >= base_of_stack) return(ENOMEM);   /* sp too high */

  /* compute size of gap between stack and data segments. */
  delta = (long) stack_vir - (long) sp_clicks;
  lower = (delta > 0 ? sp_clicks : stack_vir);

#define SAFETY_BYTES  (384 * sizeof(char *))
#define SAFETY_CLICKS ((SAFETY_BYTES + CLICK_SIZE - 1) / CLICK_SIZE)
  gap_base = data_vir + new_clicks + SAFETY_CLICKS;
  if (lower < gap_base) return(ENOMEM); /* data and stack collided */

  /* update data length (but not data origin) on behalf of brk() system call. */
  if (new_clicks != data_len) {
    data_len = new_clicks;
    changed |= DATA_CHANGED;
  }

  /* Update stack length and origin due to change in stack pointer. */
```

```
  if (delta > 0) {
    stack_vir -= delta;
    stack_phy -= delta;
    stack_len += delta;
    changed |= STACK_CHANGED;
  }

  if(changed & STACK_CHANGED){
    EXEC SQL UPDATE segment
      SET virtual_address = :stack_vir,
          physical_address = :stack_phy,
          length = :stack_len
    WHERE seg_id = :stack_seg;
  }

  if(changed &  DATA_CHANGED) {
    EXEC SQL UPDATE segment
      SET length = :data_len
    WHERE seg_id = :data_seg;
  }

  return(OK);
}
```

A.3.4 MINIX transaction types

Finite state machines

The finite state machines for the system calls are shown in Figure A.7 to A.11. The number attached to each state is the state identifier, and the text inside a state box is the data item accessed. In addition, the **r** or **w** attached to each state indicates whether the state read or write the data item. We include in Table A.1 to A.5 the probabilities of transitions from the states with multiple successors. The state identifiers in the tables are from Figure A.7 to A.11.

Calculating the state costs

We assume the general features of the kernel data structures are:

hole: **hole** is an array of **hole_t** elements of the size 128, which specifies the maximal number of holes. However, the real hole list is chained by the **h_next** pointer field

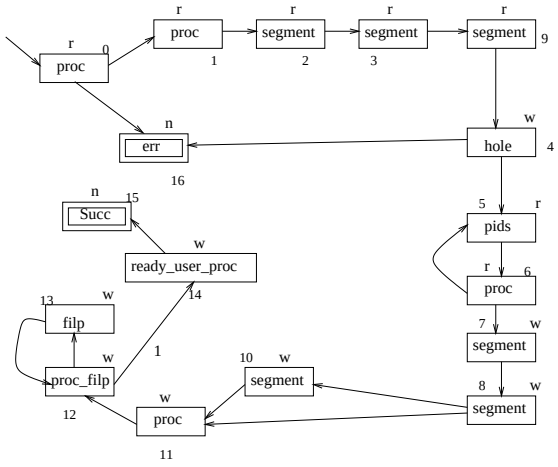


Figure A.7: The finite state machine of **fork**

arc	*prob*	*arc*	*prob*	*arc*	*prob*
$(0, 1)$	0.95	$(0, 16)$	0.05	$(4, 5)$	0.95
$(4, 16)$	0.05	$(6, 5)$	0.967	$(6, 7)$	0.033
$(8, 11)$	0.67	$(8, 10)$	0.33	$(12, 13)$	0.95
$(12, 14)$	0.05				

Table A.1: The probabilities of state transitions in **fork**

arc	*prob*	*arc*	*prob*	*arc*	*prob*
$(1, 2)$	0.95	$(1, 3)$	0.05	$(3, 4)$	0.95
$(3, 5)$	0.05	$(10, 10)$	0.9375	$(10, 11)$	0.0417
$(10, 19)$	0.0208	$(11, 12)$	0.2	$(11, 13)$	0.8
$(12, 14)$	0.8	$(12, 15)$	0.2	$(21, 22)$	0.2
$(21, 23)$	0.8	$(22, 24)$	0.8	$(22, 25)$	0.2
$(30, 31)$	0.67	$(30, 32)$	0.33	$(32, 36)$	0.5
$(32, 34)$	0.125	$(32, 33)$	0.375	$(37, 38)$	0.5
$(37, 40)$	0.5				

Table A.2: The probabilities of state transitions in **exit**

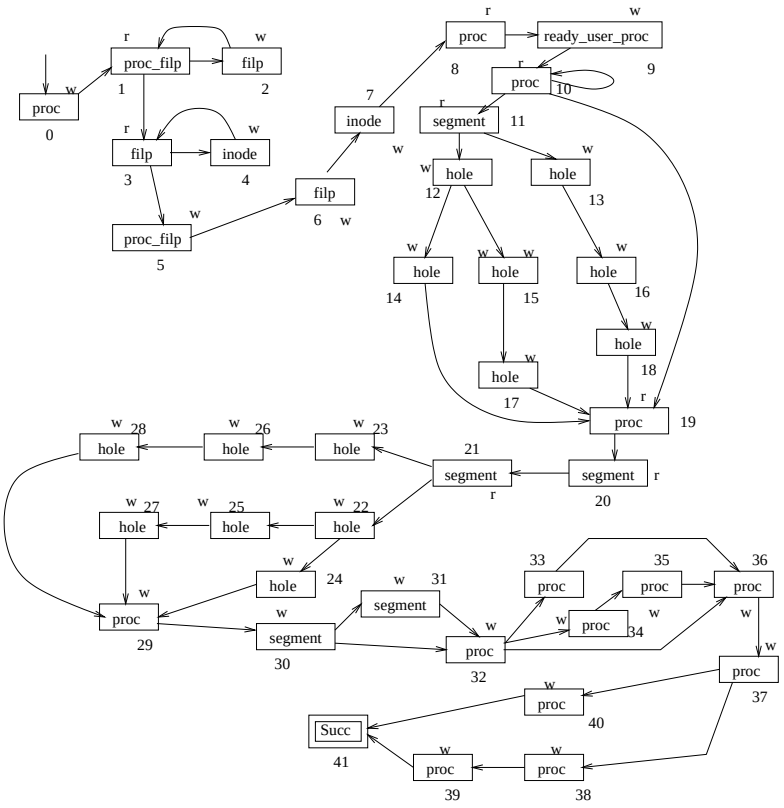


Figure A.8: The finite state machine of **exit**

arc	*prob*	*arc*	*prob*	*arc*	*prob*
$(0, 1)$	0.225	$(0, 3)$	0.05	$(0, 12)$	0.225
$(0, 5)$	0.225	$(0, 6)$	0.05	$(0, 13)$	0.225

Table A.3: The probabilities of state transitions in **waitpid**

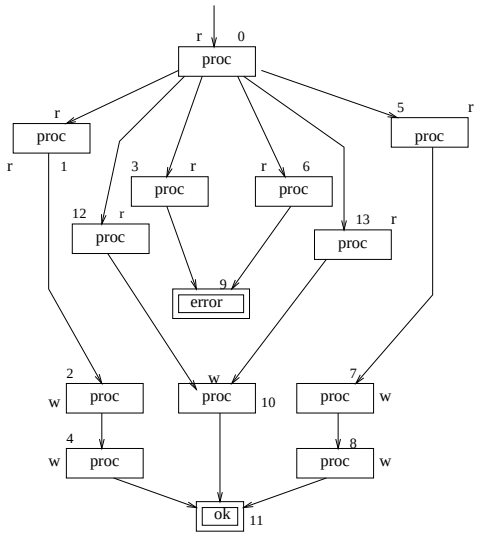


Figure A.9: The finite state machine of **waitpid**

arc	*prob*	*arc*	*prob*	*arc*	*prob*
$(4, 4)$	0.9375	$(4, 5)$	0.0575	$(4, 6)$	0.005
$(5, 6)$	0.005	$(5, 7)$	0.995	$(7, 7)$	0.9375
$(7, 8)$	0.0417	$(7, 16)$	0.0208	$(8, 9)$	0.2
$(8, 10)$	0.8	$(9, 11)$	0.8	$(9, 12)$	0.2
$(17, 18)$	0.2	$(17, 19)$	0.8	$(18, 21)$	0.2
$(18, 22)$	0.8	$(26, 27)$	0.67	$(26, 28)$	0.33
$(28, 29)$	0.67	$(28, 30)$	0.33	$(33, 34)$	0.95
$(35, 36)$	0.95	$(35, 37)$	0.05		

Table A.4: The probabilities of state transitions in **exec**

arc	*prob*	*arc*	*prob*	*arc*	*prob*
$(1, 2)$	0.05	$(1, 3)$	0.95	$(3, 2)$	0.05
$(3, 4)$	0.095	$(3, 7)$	0.19	$(3, 5)$	0.19
$(3, 6)$	0.475				

Table A.5: The probabilities of state transitions in **brk**

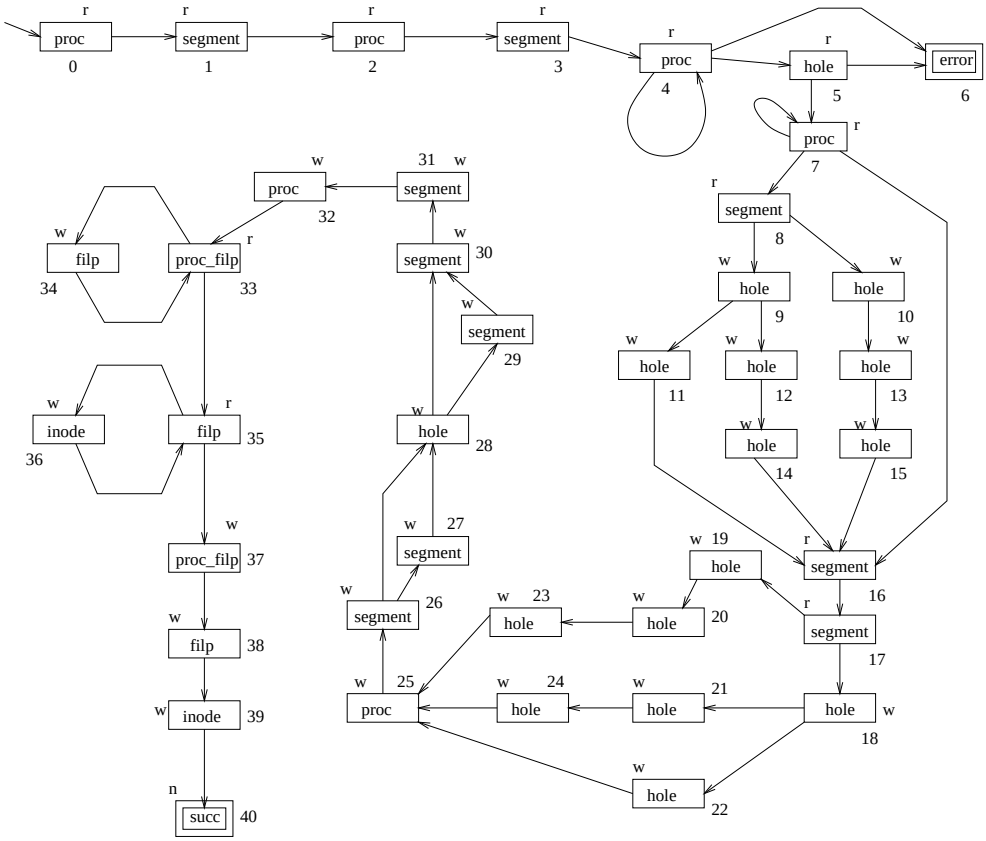


Figure A.10: The finite state machine of **exec**

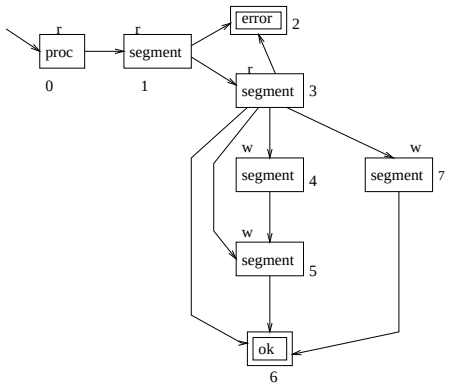


Figure A.11: The finite state machine of **brk**

in the **hole_t** data structure. We assume that on average the hole list has $\frac{128}{2} = 64$ elements. For segment allocation, we assume on average the searching goes over $\frac{64}{2} = 32$ elements before it finds a hole big enough for the segment.

proc: **proc** is an array of **proc_t** elements of size 32. Whether the element is a valid is indicated by the lowest bit (**IN_USE**) in the **proc_flags** field of **proc_t**. We assume that $\frac{3}{4} \times 32 = 24$ elements in the array are in use. We also assume:

- Given a process id, on average a searching scans $\frac{32}{2} = 16$ elements in the array to find the corresponding process descriptor.
- On average there are $32 - 24 = 8$ **proc_t** elements are not in use. They are distributed uniformly in the array. To find the first free element in the **proc** array, the searching scans on average $\frac{32}{8} = 4$ elements.

rdy_head: We assume the probability that the ready queue of user-level processes (**rdy_head[USER_Q]**) is empty is $\frac{1}{10}$, and the respective probabilities that the ready queue of kernel tasks (**rdy_head[TASK_Q]**) and servers (**rdy_head[SERVER_Q]**) are $\frac{1}{4}$.

filp: The descriptors of the files opened by a process is represented by the **fp_filp** array enclosed in the **proc_t**. Each element of the array is a pointer to **filp_t**, and the size

of the array is 20. We assume that on average a process open $\frac{20}{2} = 10$ files, so 10 pointers in the array are not NULL.

We got the costs manually. We illustrate how to get the costs by examples.

Example A.3.1. *In **fork**, we need to get the information of the parent process (state* 1 *in Figure A.7). The SQL statement embedded in the C/SQL program (Appendix A.3.3) is:*

```
EXEC SQL SELECT
  parent_pid into :parent_pid,
  wait_pid into :wait_pid,
  exit_status into :exit_status,
  sig_status into :sig_status,
  data_segment into :data_segment,
  text_segment into :text_segment,
  stack_segment into :stack_segment,
  proc_flags into :proc_flags,
  ino into :ino,
  dev into :dev,
  ctime into :ctime
FROM proc
WHERE pid = :current_pid;
```

The C statements that process the query, taken from Appendix A.3.1, are listed with line numbers:

```
/*line 1*/   for(i=0; i<NR_PROCS; i++){
/*line 2*/     if((proc[i].proc_flags & IN_USE != 0) &&
/*line 3*/        (proc[i].pid == parent_pid))
/*line 4*/     {
/*line 5*/       parent_proc = proc[i];
/*line 6*/       break;
/*line 7*/     }
/*line 8*/   }
/*line 9*/   ino = parent_proc.ino;
/*line 10*/  dev = parent_proc.dev;
/*line 11*/  ctime = parent_proc.ctime;
/*line 12*/  sp = parent_proc.sp;
/*line 13*/  wpid = parent_proc.wpid;
/*line 14*/  proc_flags = parent_proc.proc_flags;
```

The corresponding assembly instructions for the above C code are shown below. They are annotated with the line numbers in the C program.

```
!*********************************
!********** line 1***************
!*********************************
        movl    $0, -12(%ebp)
```

```
!1 instruction
!**********************************
.L8:
        cmpl    $31, -12(%ebp)
        jle     .L11
!2 instructions
!**********************************
        jmp     .L9
!1 instruction
!**********************************
!************line 2 ***************
!**********************************
.L11:
        movl    -12(%ebp), %edx
        movl    %edx, %eax
        sall    $3, %eax
        addl    %edx, %eax
        sall    %eax
        addl    %edx, %eax
        sall    $3, %eax
        movl    proc+60(%eax), %eax
        andl    $1, %eax
        testl   %eax, %eax
        je      .L10
!11 instructions
!**********************************
!************line 3 ***************
!**********************************
        movl    -12(%ebp), %edx
        movl    %edx, %eax
        sall    $3, %eax
        addl    %edx, %eax
        sall    %eax
        addl    %edx, %eax
        sall    $3, %eax
        movl    proc(%eax), %eax
        cmpl    8(%ebp), %eax
        jne     .L10
!10 instructions
!**********************************
!************line 5***************
!**********************************
        movl    -12(%ebp), %edx
        movl    %edx, %eax
        sall    $3, %eax
        addl    %edx, %eax
        sall    %eax
        addl    %edx, %eax
```

```
        sall    $3, %eax
        leal    -200(%ebp), %edi
        leal    proc(%eax), %esi
        cld
        movl    $38, %eax
        movl    %eax, %ecx
        rep
        movsl
!14 instructions
!*********************************
!************line 6***************
!*********************************
        jmp     .L9
!1 instruction
!*********************************
!************line 8***************
!*********************************
.L10:
        leal    -12(%ebp), %eax
        incl    (%eax)
        jmp     .L8
!3 instructions
!*********************************
!*********************************
!************line 9***************
.L9:
        movl    -148(%ebp), %eax
        movw    %ax, -222(%ebp)
!2 instructions
!*********************************
!************line 10**************
!*********************************
        movw    -146(%ebp), %ax
        movw    %ax, -224(%ebp)
!2 instructions
!*********************************
!************line 11**************
!*********************************
        movl    -144(%ebp), %eax
        movl    %eax, -228(%ebp)
!2 instructions
!*********************************
!************line 12**************
!*********************************
        movl    -136(%ebp), %eax
        movl    %eax, -240(%ebp)
!2 instructions
!*********************************
```

```
!************line 13***************
!*********************************
        movl    -192(%ebp), %eax
        movl    %eax, -236(%ebp)
!2 instructions
!*********************************
!************line 14***************
!*********************************
        movl    -140(%ebp), %eax
        movl    %eax, -232(%ebp)
!2 instructions
!*********************************
```

NR_PROC *is* 32. *By our assumption,* $\frac{3}{4}$ *of the elements in* ***proc*** *are in use and they are distributed evenly. In addition, the loop iterates* $\frac{32}{2} = 16$ *times on average to find the qualified element that matches the process id parameter. Therefore, from the the number of assembly instructions for each each line of the C code, the cost to process the SQL statement is*

$$\begin{aligned} cost =& 1 + 2 \times 16 && \textit{(line 1)} \\ &+ 11 \times 16 && \textit{(line 2)} \\ &+ 10 \times 16 \times \frac{3}{4} && \textit{(line 3)} \\ &+ 14 && \textit{(line 5)} \\ &+ 1 && \textit{(line 6)} \\ &+ 3 \times (16 - 1) && \textit{(line 8)} \\ &+ 2 + 2 + 2 + 2 + 2 + 2 && \textit{(line 9-14)} \\ =& 401 \end{aligned}$$

Example A.3.2. *In* ***exit****, the opened files are to be closed. The C/SQL statements (from Appendix A.3.3) are:*

```
EXEC SQL UPDATE filp set filp_count = filp_count - 1
WHERE EXISTS (
  SELECT * FROM proc_filp
  WHERE proc_filp.proc_id = :current_pid
  AND   proc_filp.filp_id = filp.filp_id
);

EXEC SQL UPDATE inode set count = count - 1
WHERE EXISTS (
```

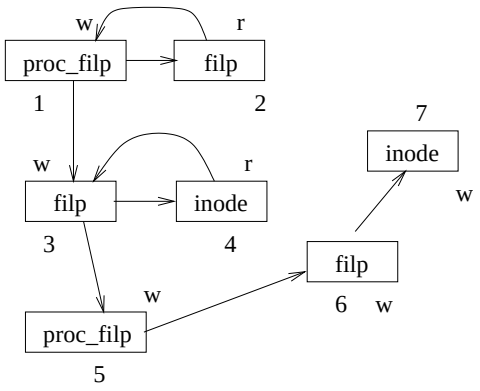


Figure A.12: Part of the finite state machine of **exit**

```
  SELECT * from filp
  WHERE filp.filp_count = 0
  AND filp.inode_id = inode.inode
)

EXEC SQL DELETE FROM proc_filp
WHERE proc_id = :current_pid;

EXEC SQL DELETE FROM filp
WHERE filp_count = 0;

EXEC  SQL DELETE FROM inode
WHERE count = 0;
```

They represent the subgraph of the finite state in Figure A.8 generated by Node 1 *to* 7*. We show the subgraph in Figure A.12.*

The C program that process the queries, taken from Appendix A.3.3, are:

```
/*line 1*/   for(i=0; i<OPEN_MAX; i++){
/*line 2*/     if((current_proc_ptr->fp_filp[i])!=NIL_FILP){
/*line 3*/       (current_proc_ptr->fp_filp[i])->filp_count--;
/*line 4*/       if((current_proc_ptr->fp_filp[i])->filp_count == 0){
/*line 5*/         (current_proc_ptr->fp_filp[i])->filp_ino->count--;
/*line 6*/       }
/*line 7*/       current_proc_ptr->fp_filp[i] = NIL_FILP;
/*line 8*/     }
/*line 9*/   }
```

To map the C statements to the embedded SQL statements, we partition the ***for*** *loop. The*

first embedded SQL statement that joins ***filp*** *and* ***proc_filp*** *is processed by scanning the* ***fp_filp*** *array in the process descriptor. So the first join (the loop with with state* 1 *and* 2 *in Figure A.12) is processed by line* 1, 2, 3, 8, *and* 9. *The second one that joins* ***filp*** *and* ***inode*** *is processed by line* 1, 2, 5, 6, 8, *and* 9. *The third deletes the relevant rows in the* ***proc_filp*** *table. This is processed by line* 1, 2, 7, 8, 9. *The last two* ***delete*** *statements actually have nothing to do at the physical level, because when the reference counts in* ***inode_t*** *and* ***filp_t*** *equal to* 0, *the entry is free.*

The assembly code is:

```
!**********************************
!************line 1****************
!**********************************
        movl    $0, -4(%ebp)
!1 instruction
!**********************************
.L7:
        cmpl    $19, -4(%ebp)
        jle     .L10
!2 instructions
!**********************************
        jmp     .L8
!1 instruction
!**********************************
!************line 2****************
!**********************************
.L10:
        movl    -12(%ebp), %edx
        movl    -4(%ebp), %eax
        cmpl    $0, 68(%edx,%eax,4)
        je      .L9
!4 instructions
!**********************************
!************line 3****************
!**********************************
        movl    -12(%ebp), %edx
        movl    -4(%ebp), %eax
        movl    68(%edx,%eax,4), %eax
        decl    (%eax)
!4 instructions
!**********************************
!************line 4****************
!**********************************
        movl    -12(%ebp), %edx
        movl    -4(%ebp), %eax
        movl    68(%edx,%eax,4), %eax
```

```
        cmpl    $0, (%eax)
        jne     .L12
!5 instructions
!*********************************
!************line 5***************
!*********************************
        movl    -12(%ebp), %edx
        movl    -4(%ebp), %eax
        movl    68(%edx,%eax,4), %eax
        movl    4(%eax), %eax
        decl    (%eax)
!5 instructions
!*********************************
!************line 7***************
!*********************************
.L12:
        movl    -12(%ebp), %edx
        movl    -4(%ebp), %eax
        movl    $0, 68(%edx,%eax,4)
!3 instructions
!*********************************
!************line 9***************
!*********************************
.L9:
        leal    -4(%ebp), %eax
        incl    (%eax)
        jmp     .L7
!3 instructions
!*********************************
```

The costs are:

State 1 *is processed by the 2 instructions immediately under* ***.L7*** *for line* 1*, i.e., one* ***cmpl*** *and one* ***jle****. Its cost is* 2*.*

State 2 *is processed by line* 2 *and* 3*. As we assume half of the elements in* ***fp_filp*** *are non-null pointers and the other half are null, there is* 50% *probability that line* 3 *is executed. Therefore,*

$$\begin{aligned} cost =& 4 && \textit{(line 2)} \\ &+ 4 \times 0.5 && \textit{(line 3)} \\ =& 6 \end{aligned}$$

Arc(1, 2) *has cost* 0*.*

Arc(2, 1) *is done by line* 9. *Its cost is* 3.

Arc(1, 3) *is processed by* 2 *instructions for line* 1 *(the first one* ***movl*** *that initializes and the last one* ***jmp****). Its cost is* 2.

State 3 *is similar to state* 1. *Its cost is* 2.

Arc(3, 4) *has cost* 0.

State 4 *is processed by line* 2, 5, *and* 6 *(empty). As we assume that half of* ***fp_filp*** *elements are not null, the probability that line* 4 *is executed is* 50%. *We also assume the probability that line* 5 *is executed is* 40%.

$$\begin{aligned} cost =& 4 && \textit{(line 2)} \\ &+ 5 \times 0.5 && \textit{(line 4)} \\ &+ 5 \times 0.4 && \textit{(line 5)} \\ =& 9 \end{aligned}$$

Arc(4, 3) *is processed by line* 9. *The cost is* 3.

State 5 *is processed by line* 1, 2, 7, 8 *(empty), and* 9. *As the state occupies the whole loop block,* ***OPEN_MAX*** *is* 20, *and we assume* 10 *elements are not null, the total cost is,*

$$\begin{aligned} cost =& 1 + 2 \times 20 + 1 && \textit{(line 2)} \\ &+ 4 \times 20 && \textit{(line 2)} \\ &+ 3 \times 10 && \textit{(line 7)} \\ &+ 3 \times 19 && \textit{(line 9)} \\ =& 209 \end{aligned}$$

Arc(3, 5) *is done by the last instruction for line* 1. *So the cost is* 1.

Arc(5, 6), State 6 **and State** 7 *all have the cost* 0.

The transaction types with the costs we have acquired are shown in Figure A.13 to A.17. The numbers attached to the states and the arcs are their costs.

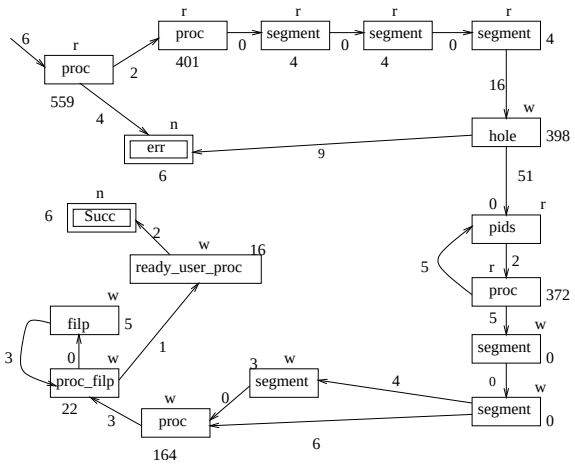


Figure A.13: The costs of **fork**

A.3.5 Calculating the locking cost

Spinlock

We use the assembly code that implements spinlock locking in Linux [11]. It is assumed that the lock is encoded in a byte **slp**.

```
1:  lock; decb slp
    jns 3f
2:  cmpb $0, slp
    pause
    jle 2b
    jmp 1b
3:
```

The unlocking code is:

```
lock; movb $1, slp
```

So when the value of **slp** is 1, the lock is free. The locking code will directly jump to label 3. So it takes 2 instructions (the first 2 instructions). Otherwise, the code loops after label 2 and tests whether the value is reset to 1. If it is, the code goes back to label 1 to retest whether the lock is still free. So the resumption of the process takes 4 instructions, including a **jle** and a **jmp**. The unlocking has only 1 instruction.

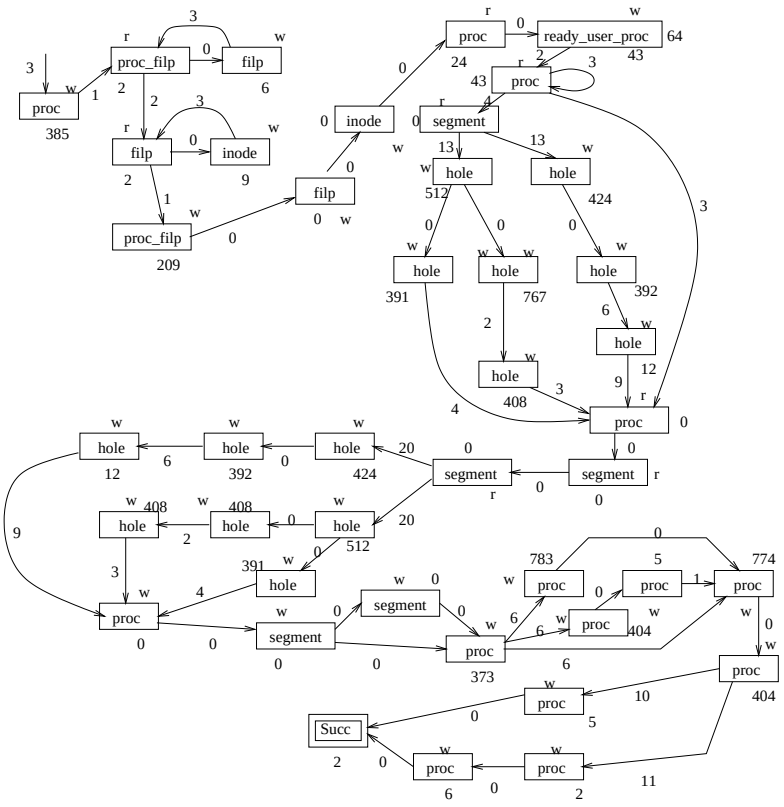


Figure A.14: The costs of **exit**

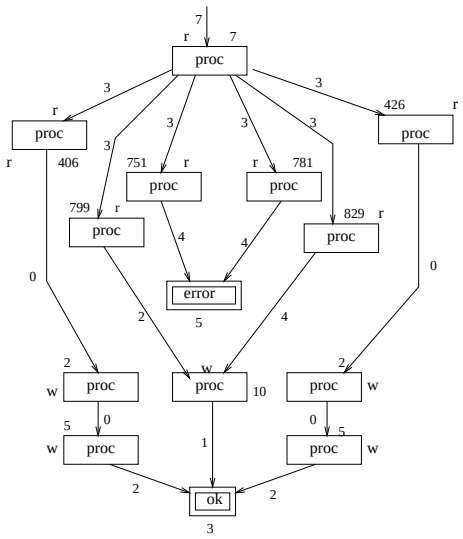


Figure A.15: The costs of **waitpid**

X-lock

First we consider the costs of x-locks. The data structures of an x-lock are:

```
#define LOCK_FREE 0
#define LOCK_X    1

typedef int lock_mode_t;

/* the waiting queue */
struct wait_queue_t {
  struct proc_t * head;
  struct proc_t * tail;
};

/* the semaphore */
struct semaphore_t {
  lock_mode_t mode;      /* free or exclusively locked */
  spinlock_t  lock;
  pid_t       owner;
  struct wait_queue_t wait_queue;
};
```

Also we add a pointer field into the process descriptor structure (**proc_t** defined in Appendix A.3.1.

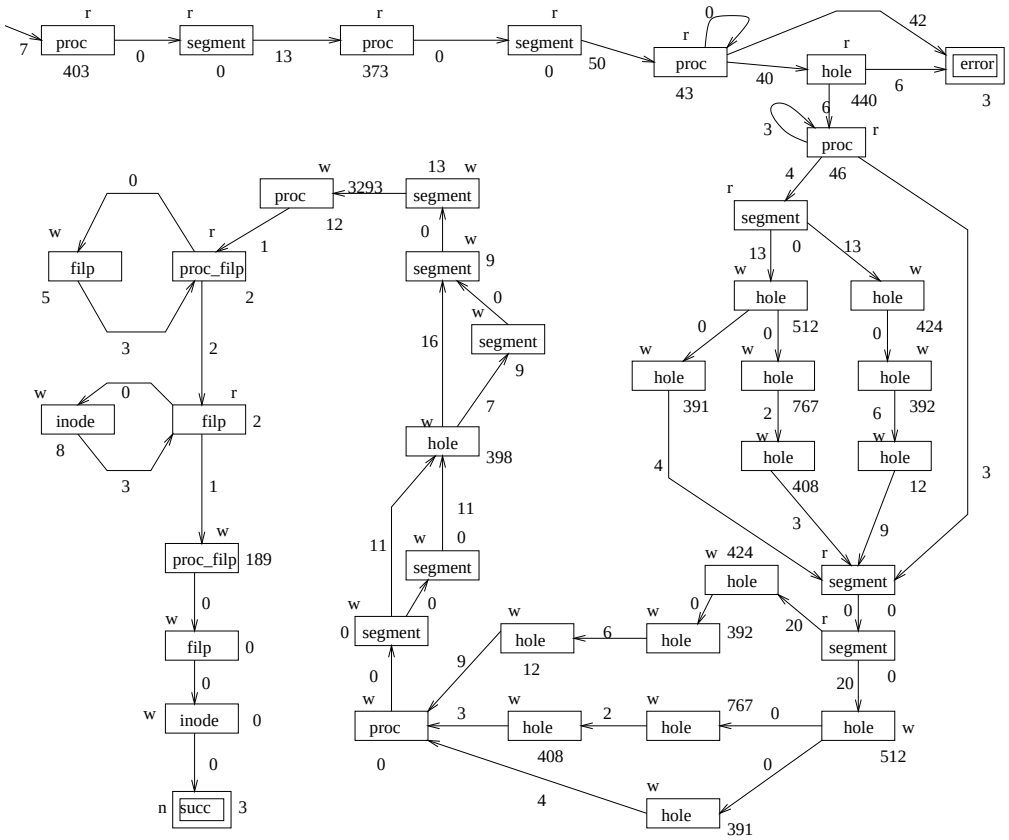


Figure A.16: The costs of **exec**

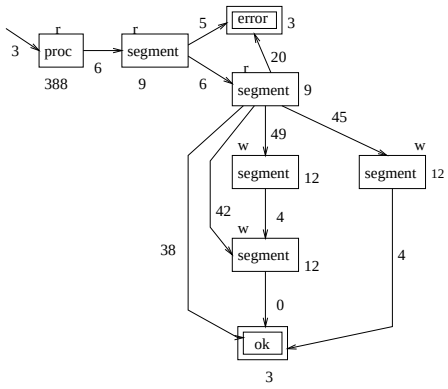


Figure A.17: The costs of **brk**

```
struct proc_t {
   .....
  /* add waiting queue list */
  struct proc_t * wait_next;
} proc[NR_PROCS];
```

The pointer links the process descriptor into the waiting queue. Based on the Linux spinlock, we implement the x-lock locking and unlocking by ourselves. The code is given below.

```
#define INVALID_PID (-1000)

/* add the process to the tail of the waiting queue */
void add_wait(struct wait_queue_t *wait_queue, struct proc_t *proc)
{
  proc->wait_next = NIL_PROC;

  /* add to tail */
  if(wait_queue->tail){
    (wait_queue->tail)->wait_next = proc;
  }
  wait_queue->tail = proc;

  /* add to header if the queue is empty */
  if(!(wait_queue->head)){
    (wait_queue)->head = proc;
  }
}
```

```
/* remove the process from the ready list,
   and schedule another process to run. */
void unschedule(struct proc_t * current_proc_ptr){
  struct proc_t *xp, *rp, **qtail;
    /* copied from unready() and pick_proc() */
  if((xp = rdy_head[USER_Q]) == current_proc_ptr){
    rdy_head[USER_Q] = xp->p_next_ready;

    /* pick the next proc to run */
    if ( (rp = rdy_head[TASK_Q]) != NIL_PROC) {
        proc_ptr = rp;
    }
    else
    if ( (rp = rdy_head[SERVER_Q]) != NIL_PROC) {
       proc_ptr = rp;
    }
    else
    if ( (rp = rdy_head[USER_Q]) != NIL_PROC) {
        proc_ptr = rp;
        bill_ptr = rp;
    }else{
    /* No one is ready.  Run the idle task.  The idle task might be made an
     * always-ready user task to avoid this special case.
     */
      bill_ptr = proc_ptr = &proc[IDLE];
    }
  }
  qtail = &rdy_tail[USER_Q];

  while (xp->p_next_ready != current_proc_ptr){
    if ( (xp = xp->p_next_ready) == NIL_PROC) break;
  }

  if(xp != NIL_PROC){
    xp->p_next_ready = xp->p_next_ready->p_next_ready;
    if (*qtail == current_proc_ptr) *qtail = xp;
  }
}

/* acquire an exclusive lock. */
void x_lock(struct proc_t* current_proc, struct semaphore_t * sem){
  spin_lock(&(sem->lock));

  if(sem->mode != LOCK_X) {
     sem->mode = LOCK_X;
     sem->owner = current_proc->pid;
     spin_unlock(&(sem->lock));
```

```
    return;
  }
  else{
    /* must be locked */

    /* if the current proc already locks it, do nothing. */
    if(sem->owner == current_proc->pid){
      spin_unlock(&(sem->lock));
      return;
    }

    /* block it. */
    add_wait(&(sem->wait_queue),current_proc);
    unschedule(current_proc);
    spin_unlock(&(sem->lock));
  }
}

/* remove the process from the head of the waiting queue */
void remove_wait(struct wait_queue_t *queue, struct proc_t *proc){
  proc = NIL_PROC;
  if(queue->head != NIL_PROC){
    proc = queue->head;
    queue->head = (queue->head)->wait_next;
  }
}

/* put the process to the ready list */
void schedule(struct proc_t *current_proc){
  if(rdy_head[USER_Q] == NIL_PROC)
    rdy_tail[USER_Q] = current_proc;
  rdy_head[USER_Q] -> p_next_ready = rdy_head[USER_Q];
  rdy_head[USER_Q] = current_proc;
}

/* unlock */
void x_unlock(struct proc_t *current_proc, struct semaphore_t *sem){
  struct proc_t *proc_wakeup;

  spin_lock(&(sem->lock));
  /* I am not the owner of the lock. do nothing. */
  if(sem->mode != LOCK_X || sem->owner != current_proc->pid){
    spin_unlock(&(sem->lock));
    return;
  }

  /* wake up the first proc in the wait queue. */
  remove_wait(&(sem->wait_queue), proc_wakeup);
```

```
  if(NIL_PROC != proc_wakeup){
    /* if wait queue is not empty */
    schedule(proc_wakeup);
    sem->owner =  proc_wakeup->pid;
  }
  else{
    /* no waiting process */
    sem->owner = INVALID_PID;
    sem->mode = LOCK_FREE;
  }

  spin_unlock(&(sem->lock));
}
```

By using the machine instruction counting method used in Section 4.2.2, we calculate the average cost for locking. We assume the probability that queue is empty is $\frac{1}{2}$. We omit further details and just give the result. For **locking**, if the lock is free and the process can get it immediately, the cost is 30 instructions; if the lock is held by another process and the process has to wait, the cost is 134. The **unlocking** takes 62 instructions.

Rw-lock

The second type of lock, rw-lock, allows readers to share it. It follows the classical model of multiple readers and single writer. The data structures are:

```
#define LOCK_INVALID (-1)
#define LOCK_FREE 0
#define LOCK_S    1
#define LOCK_X    2

typedef int lock_mode_t;

/* the waiting queue */
struct wait_queue_t {
  struct proc_t * head;
  struct proc_t * tail;
};

struct rw_semaphore_t {
  lock_mode_t mode;     /* free, share, or or exclusive */
  spinlock_t  lock;
  int         count;    /* number of holders: 0 free,
                           1 exclusive or shared, >1 shared */
  struct wait_queue_t wait_queue;
```

```
};
```

We add three fields to the process descriptor structure (**proc_t** originally defined in Appendix A.3.1.

```
struct proc_t {
   .....
  struct proc_t * wait_next; /* waiting queue link */
  lock_mode_t hold_lock;     /* lock mode the process is holding */
  lock_mode_t wait_for_lock; /* lock mode the process is waiting for */
} proc[NR_PROCS];
```

wait_next is the linking pointer inside the waiting queue. **hold_lock** shows the mode of the lock the process is holding (free, shared, or exclusive). When a process is waiting for a lock in the waiting queue, **wait_for_lock** shows the mode of the lock it is waiting for. The function **schedule**, **unschedule**, **add_wait**, and **remove_wait** are the same as in Appendix A.3.5. The other functions added are listed below.

```
/* check what kind of lock the first process in the wait queue
   requests for */
lock_mode_t get_header_wait_for(struct wait_queue_t *queue){
  if(queue->head == NIL_PROC){
    return LOCK_INVALID;
  }
  else{
    return queue->head->wait_for_lock;
  }
}

/* locking */
void rw_lock(struct proc_t         *current_proc,
             struct rw_semaphore_t *sem,
             lock_mode_t            mode)
{
  spin_lock(&(sem->lock));
  if(sem->mode == LOCK_FREE) {

     /* lock free, so obtain it. */
     sem->mode = mode;
     sem->count = 1;
     current_proc->hold_lock = mode;
     spin_unlock(&(sem->lock));
     return;
  }
  else{
     /*
     /* if the current proc already locks it, do nothing. */
```

```
    if(mode <= current_proc->hold_lock){
      spin_unlock(&(sem->lock));
      return;
    }
    else
    if(mode == LOCK_S && sem->mode == LOCK_S &&
       (sem->wait_queue).head == NIL_PROC)
    {
      sem->mode = mode;
      sem->count++;
      current_proc->hold_lock = mode;
      spin_unlock(&(sem->lock));
      return;
    }

    /* block it. */
    add_wait(&(sem->wait_queue),current_proc);
    current_proc->wait_for_lock = mode;
    unschedule(current_proc);
    spin_unlock(&(sem->lock));
  }
}

/* unlocking */
void rw_unlock(struct proc_t *current_proc,
               struct rw_semaphore_t *sem){
  struct proc_t *proc_wakeup;
  spin_lock(&(sem->lock));
  /* I am not the owner of the lock. do nothing. */
  if(sem->mode == LOCK_FREE ||
     sem->mode != current_proc->hold_lock) {
    spin_unlock(&(sem->lock));
    return;
  }

  sem->count--;

  /* wake up procs in the wait queue. */
  if(sem->count == 0){
    remove_wait(&(sem->wait_queue), proc_wakeup);

    /* the wait queue is empty? */
    if(NIL_PROC != proc_wakeup){
      /* not empty, wake up the first one */
      sem->mode = proc_wakeup->wait_for_lock;
      proc_wakeup->hold_lock = sem->mode;
      schedule(proc_wakeup);
```

```
      /* wake up more readers if necessary */
      if(LOCK_S == sem->mode){
        while(get_header_wait_for(&(sem->wait_queue)) == LOCK_S)
        {
          remove_wait(&(sem->wait_queue), proc_wakeup);
          proc_wakeup->hold_lock = LOCK_S;
          schedule(proc_wakeup);
        }
      }
    }
    else{
      /* wait queue empty */
      sem->mode = LOCK_FREE;
    }
  }
  current_proc->hold_lock = LOCK_FREE;
  spin_unlock(&(sem->lock));
}
```

Following the same assumption as the above x-lock, we calculate the numbers of instructions of locking and unlocking. For the **locking** in shared mode, a successful locking takes 37 instructions, and if the lock cannot be granted, a blocking takes 136 instruction. For the **locking** in exclusive mode, the corresponding costs are 32 and 131 respectively. **Unlocking** takes 68 instructions on average.

Bibliography

[1] Linux Cross-Reference. URL http://lxr.linux.no.

[2] Transaction Processing Performance Council. URL http://www.tpc.org.

[3] Data Management in Embedded Real-Time Software Systems. NSERC Colloborative Research Project 661-135/94, Government of Canada, 1994.

[4] Static Embedded OQL for Real-Time Applications. Research Project, Information Technology Research Center, Government of Ontario, 1996.

[5] Rudolf Bayer and Mario Schkolnick. Concurrency of Operations on B-Trees. *Acta Informatica*, 9:1–21, 1977.

[6] C. Beeri, P.A. Bernstein, N. Goodman, M.Y. Lai, and D.E. Shasha. A Concurrency Control Theory for Nested Transactions (Preliminary Report). In *Proceedings of the Second ACM Symposium on Principles of Distributed Computing*, pages 45–62, Montreal Canada, August 1983.

[7] Catriel Beeri, Philip A. Bernstein, and Nathan Goodman. A Model for Concurrency in Nested Transactions Systems. *Journal of ACM*, 36(2):230–269, April 1989.

[8] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[9] Stephen Blott and Henry F. Korth. An almost-serial protocol for transaction execution in main-memory database systems. In *Proceedings of the 28th VLDB Conference*, 2002.

[10] Alex Borgida. Description Logics in Data Management. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):671–682, October 1995.

[11] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2nd edition, 2002.

[12] Gael N. Buckley and Avi Silberschatz. Concurrency Control in Graph Protocols by Using Edge Locks. In *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 45–50, 1984.

[13] Petrus Kai Chung Chan. Optimizing OQL on Legacy Main Memory Data Structures with Existential Graphs. Master's thesis, University of Waterloo, Waterloo, ON, Canada, 1997.

[14] Vinay K. Chaudhri and Vassos Hadzilacos. Safe Locking Policies for Dynamic Databases. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California*, pages 233–244. ACM Press, 1995.

[15] Vinay K. Chaudhri, Vassos Hadzilacos, and John Mylopoulos. Concurrency Control for Knowledge Bases. In *Proceedings of the 3rd International Conference on Knowledge Representation and Reasoning*, pages 762–773, October 1992.

[16] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, 2001.

[17] Albert Croker and David Maier. A dynamic tree-locking protocol. In *Proceedings of the Second International Conference in Data Engineering*, pages 49–56, Los Angeles, CA, USA, 1986.

[18] K. P. Eswaran, J. Gray, R.A. Lorie, and I.L. Traiger. The Notions of Consistency and Predicate Locks in Database System. *Communications of the ACM*, 19(16):624–633, 1976.

[19] Pasal Felber and Michael K. Reiter. Advanced Concurrency Control in Java. *Concurrency and Computation: Practice and Experience*, 14(4):261–285, 2002.

[20] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of the ACM SIGMOD Annual Conference on Management of Data*, pages 249–259, New York, NY, USA, 1987. ACM Press.

[21] Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(2):509–516, December 1992.

[22] Seymour Ginsburg and Richard Hull. Order Dependency in the Relational Model. *Theoretical Computer Science*, 26:149–195, 1983.

[23] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[24] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991.

[25] Z. Kedem, C. Mohan, and A. Silberschatz. An efficient deadlock removal scheme for non-two-phase locking protocols. In *Proceedings of 8th International Conference on Very Large Data Bases*, pages 91–97, Mexico City, Mexico, September 1982.

[26] Zvi Kedem and Abraham Silberschatz. Controlling Concurrency Using Locking Protocols (Preliminary Report). In *Proceedings of 20th Symposium on Foundations of Computer Science*, pages 274–285, October 1979.

[27] Zvi M. Kedem and Abraham Silberschatz. Locking Protocols: From Exclusive to Shared Locks. *Journal of ACM*, 30(4), October 1983.

[28] Udo Kelter. The queue protocol: A deadlock-free homogeneous non-two-phase locking protocol. In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 142–151, Austin, Texas, USA, March 1988.

[29] H. T. Kung and Philip L. Lehman. Concurrency Manipulation of Binary Search Trees. *ACM Transactions on Database Systems (TODS)*, 5(3):354–382, September 1980.

[30] Yat-Sang Kwong and Derick Wood. A New Method for Concurrency in B-Trees. *IEEE Transactions on Software Engineering*, 8(3):211–222, May 1982.

[31] Stella Yuen Mei Lai. On Integrating Legacy Main Memory Data Structure with Database Schema. Master's thesis, University of Waterloo, Waterloo, ON, Canada, 1999.

[32] Vladimir Lanin and Dennis Shasha. Tree Locking on Changing Trees. Technical report, Courant Institute of Mathematical Sciences, New York University, 1990.

[33] Philip L. Lehman and S. Bing Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems (TODS)*, 6(4):650–670, December 1981.

[34] Huizhu Liu, David Toman, and Grant Weddell. Fine Grained Information Integration with Description Logics. In *Proceedings of the 2002 International Description Logics Workshop (DL-2002)*, pages 1–12, Toulouse, France, April 2002.

[35] Udi Manbar and Richard E. Ladner. Concurrency control in a dynamic search structure. *ACM Transactions on Database Systems (TODS)*, 9(3):439–455, September 1984.

[36] C. Mohan, Donald Fussell, and Zvi M. Kedem. Locking Conversion in Non-Two-Phase Locking Protocols. *IEEE Transactions on Software Engineering*, 11(1), january 1985.

[37] Y. Mond and Yoav Raz. Concurrency Control in B+-Trees Databases Using Preparatory Operations. In *Proceedings of 11th International Conference on Very Large Data Bases (VLDB)*, pages 331–334, Stockholm, Sweden, August 1985.

[38] J. E.B. Moss. *Nested transactions: an approach to reliable distributed computing.* MIT Press, 1985.

[39] Don Roberts. *The Existential Graphs of Charles S. Peirce.* Mouton, The Hague, 1973.

[40] Kenneth Salem, Hector Garcia-Molina, and Jeannie Shands. Altruistic Locking. *ACM Transactions on Database Systems*, 19(1):117–165, March 1994.

[41] Dennis Shasha and Nathan Goodman. Concurrent search structure algorithms. *ACM Transactions on Database Systems (TODS)*, 13(1):53–90, March 1988.

[42] Abraham Silberschatz and Zvi Kedem. Consistency in Hierarchical Database Systems. *Journal of ACM*, 27(1), January 1980.

[43] Abraham Silberschatz and Zvi M. Kedem. A Family of Locking Protocols for Database Systems that Are Modeled by Directed Graphs. *IEEE Transactions on Software Engineering*, 8(6):558–562, November 1982.

[44] Lubomir Stanchev and Grant Weddell. Index Selection for Compiled Database Applications in Embedded Control Programs. In *Proceedings of CASCON 2002*, pages 156–170, Toronto, ON, Canada, September - October 2002.

[45] Lubomir Stanchev and Grant Weddell. Index Selection for Embedded Control Applications using Description Logics. In *Proceeding of the 2003 Internation Workshop on Description Logics (DL-2003)*, pages 9–19, Rome, Italy, September 2003.

[46] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems: design and implementation*. Prentice Hall, 2nd edition, 1997.

[47] David Toman and Grant Weddell. On Attributes, Roles, and Dependencies in Description Logics and the Ackermann Case of the Decision Problem. In Carole Goble, Deborah L. McGuinness, Ralf Möller, and Peter F. Patel-Schneider, editors, *Working Notes of the 2001 International Description Logics Workshop (DL-2001)*, pages 76–85, Stanford, CA, USA, 2001.

[48] David Toman and Grant E. Weddell. Query Processing in Embedded Control Programs. In *Proceedings of the Second International Workshop on Databases in Telecommunications*, number 2209 in Lecture Notes in Computer Science, pages 68–87. Springer-Verlag, 2001.

[49] Uresh Vahalia. *Unix Internals: the New Frontiers*. Prentice Hall, 1996.

[50] M.F. van Bommel and G.E. Weddell. Reasoning About Equations and Functional Dependencies on Complex Objects. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 6(3):455 – 469, June 1994.

[51] Grant E. Weddell. Reasoning about Functional Dependencies Generalized for Semantic Data Models. *ACM Transactions on Database Systems (TODS)*, 17(1):32–64, 1992.

[52] Gerhard Weikum. Principles and Realization Strategies of Multi-level Transaction Management. *ACM Transactions on Database Systems*, 16(1):132–180, March 1991.

[53] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann, 2001.

[54] M. Yannakakis, C. H. Papadimitriou, and H.T Kung. Locking policies: Safety and Freedom from Deadlock. In *Proceedings of 20th IEEE Symposium on Foundations of Computer Science*, pages 286–297, October 1979.

[55] Mihalis Yannakakis. A Theory of Safe Locking Policies in Database Systems. *Journal of ACM*, 29(3):718–740, July 1982.

[56] Mihalis Yannakakis. Freedom from Deadlock of Locking Policies. *SIAM Journal on Computing*, 11(2):391–408, May 1982.

[57] Heng Yu and Grant Weddell. Investigations in Tree Locking for Compiled Database Applications. Technical Report CS-2003-25, School of Computer Science, University of Waterloo, Waterloo, ON, Canada, September 2003.

[58] Heng Yu and Grant Weddell. Investigation in Tree Locking for Compiled Database Applications. In *Proceedings of CASCON 2004*, pages 217–231, Toronto, ON, Canada, October 2004.

[59] Heng Yu and Grant Weddell. Building an Embedded Control Program Workload. Technical Report CS-2005-04, School of Computer Science, University of Waterloo, Waterloo, ON, Canada, February 2005.