SQLiteOS

Vinayak Jha
Electrical Engineering and Computer Science
University of Kansas
Lawrence, Kansas, USA
Email: vinayakjha@ku.edu

Adam Podgorny
Electrical Engineering and Computer Science
Zhong Lab
University of Kansas
Lawrence, Kansas, USA
Email: apodgorny@ku.edu

***Abstract*—The standard Linux kernel provides core user space services such as interprocess communication with files. In increasingly complex operating systems and hardware environments, these core assumptions deserve reflection: Is there an alternative way of treating these core data currencies that a modern OS relies on? We propose a database-backed Linux kernel to explore this question and suggest a different philosophy. To do this, we patch the WSL2 Linux kernel to use SQLite3 and showcase our paradigm. Surgical additions to the Linux kernel allow direct implementation of our core design. In this project, we create specialized database tables, improve SQLite's virtual file system for conflict-proofing OS transactions, and implement database-backed interprocess communication, analogous to pipes in traditional POSIX. Our evaluation shows that system integrity is not compromised by such a patch, and while there are some observed slowdowns, these are currently acceptable for the present stage of development.**

1. Introduction

The traditional Unix philosophy holds that 'everything is a file.' Linux followed in this assumption, treating file descriptors and process structs as special files in system folders. The system state is described by view these files as a form of abstract "table". Specifically, process structs are held in the /proc directory, with a subdirectory for each process containing vital system statistics about that process such as ownership, page information, memory usage, network sockets, and so on.

The question may arise of synchronicity at scale. Unix and Linux began their designs assuming single core, fairly simple, devices. As devices grew more complex, as during the twin advents of multi-core devices and specialized components like GPUs, it is reasonable to question the assumptions A clear example on the Linux systems is the 'group scheduling bug' seen on the standard Linux kernel which leads to inefficient task allocation on the part of the scheduler. When we consider systems that exist remotely or with disparate parts, such as cloud system, it is plausible that alternative means of handling system process information may give both efficiency and stability benefits.

Given the ease of deployment and cross-compilation, the Windows Subsystem for Linux 2 (WSL2) [1] environment was used within Windows. This allows observation from an 'external' system, without the potentially interfering overhead of using virtual machines. Additionally, as we wanted to show generality for standard use cases, we refrained from benchmarking on devices like Raspberry Pis, which are at present not a general consumer model computer.

We have two fundamental performance indicators: stability, and execution time. Stability here means that the changes to the kernel do not compromise the system overall, and that deterministic systems do not appear to fundamentally be altered. A basic example would be crashes or kernel panics on forking or pipe opening. Less dramatically, a stale file descriptor kept in memory may mean incorrect file IO operations. For our time requirements, we aim to keep the pipe bandwidth within an order of magnitude relative to the stock WSL2 kernel's pipe bandwidth. The LMbench suite [2] is used to test both of these.

2. Prior Work

An attempt to create a Database-backed Operating system already exists. DBOS [3] promotes this concept, for many of the same reasons we offer. At time of writing, their pilot paper has the very basics implemented. The authors put forth a 3-stage program: DBOS-straw, DBOS-wood, and DBOS-brick. Their referenced 2022 paper was firmly in the 'straw' stage at time of publication. They released a progress report [4] within the year, discussing their findings. Their benchmarking does indeed suggest the feasibility of the overall concept, but it appears they are still mostly working in user space, where we intend a kernel space approach.

2.1. SQLite3

We have chosen SQLite3 [5] as our Database Management System (DBMS). SQLite3 is already in C. This means that no additional language resources are required to bring it to Linux. Additionally, SQLite is an embedded database, that is, the code is present, and is executed, in the address space of the process. We exploit this property to integrate SQLite in the Linux kernel. Furthermore, system calls would

be operating in their 'native' language, rather than requiring additional translation layers that could bog down operations.

The developers note that SQLite [6] is "aviation grade". They also guarantee ACID functionality. For this work, atomicity is a critical assumption - contention over a particular record means that non-atomic writes could corrupt the system state. Should any transaction fail, the table rolls back to the previous state.

Strong assertions are likewise given that their software has no memory leaks; Such leaks would become catastrophic over time, as unfreed memory from stale entries could pollute the memory pool to the point of inoperability.

The documentation offered also allows for confident use of their libraries and modifications of their code as the need arises; SQLite3 is highly extensible, which means it may be adapted to handling new tasks without having to involve any middleware layers.

Altogether, these factors make SQLite3 our choice for the DBMS that underpins our proposed system.

Some foundational knowledge of SQLite's implementation is necessary to understand our implementation which is expanded upon in the following subsections.

2.2. Virtual File System

SQLite has been designed as a stack of modules [7], with each module's specific implementation abstracted from other modules. The modules (top to down) are:

1) Tokenizer
2) Parser
3) Code Generator
4) Virtual Machine
5) B-Tree
6) Pager
7) OS Interface

To execute a query in SQLite, the first step is to "prepare" a query, which converts the given textual SQL query into byte code, which finally gets executed on the virtual machine. The bottom-most layer, OS Interface (or VFS), implements the interaction with the host operating system. Specifically, it implements functionalities such as opening/closing a file, along with reading and writing to the file. Additionally, it implements the locking mechanism to handle concurrency.

2.3. File Locking

To handle concurrency, SQLite implements a locking mechanism analogous to the reader/writer semaphore [8]. That is, multiple readers can be active at a time, however, only one writer can write to the database, and no readers should be active during the writing.

The OS Interface (VFS) implements the locking mechanism since the system calls are necessary to obtain locks on the file. There are 5 levels of lock, listed below, in the order of increasing exclusivity

1) None: By default, transactions begin with no lock on the database file, denoted by this lock.
2) Shared: This lock is obtained on the first read query by the transaction. Multiple shared locks can be active at a time.
3) Reserved: This lock is obtained on the first write query by the transaction. Only one transaction can acquire this lock at a time. However, new shared locks can be acquired when a reserved lock is active.
4) Pending: When a transaction performs a commit operation, the exclusive lock is requested on the database file. However, the request can be denied because other transactions hold shared locks. Hence, the transaction must wait for the shared lock to be released. In this case, the pending lock is acquired on the database file, and no new shared locks can be acquired, preventing writer starvation. Thus, this lock is transient between exclusive and reserved lock.
5) Exclusive: The exclusive lock is acquired when the database file is being written to. During an exclusive lock, no shared locks must be active. Hence, the exclusive lock is the strongest lock of all.

It is worth noting that when a non-compatible lock is requested, the VFS layer must return an error, which the application must handle. For example, if a transaction requests a reserved lock, the VFS layer will return an error if another transaction holds a reserved lock. Thus, the library will not block or wait for the lock to be available, and the application must handle the error. The application can retry the lock request at fixed intervals (for a maximum number of tries) before aborting/handling the error. While such non-determinism may be manageable in user space applications, it is not acceptable in kernel space since an incoming transaction may be updating critical information (such as user permission) or scheduling tasks. Hence, the queries in kernel space must not abort (although, non-critical queries can be aborted in future work).

Additionally, naively sleeping on errors after a lock request can lead to deadlocks. In the above example, if the newer transaction requesting the reserved lock is put to sleep, then it will hold the shared lock while sleeping. The former transaction with the reserved lock will never be able to perform a write since it will wait on the sleeping transaction to release the shared lock. We implement a solution to this problem, as described in the Implementation section.

3. Implementation

In this section, we detail our integration of SQLite3 database with the Linux kernel, and the design of the database-backed pipe interprocess communication. We first detail how the virtual file system, and locking mechanism, was extended, followed by the implementation details of database-packed pipes. For our implementation, we

used 5.15.146.1-microsoft-standard-WSL2+ kernel [1] and SQLite 3.45.1 [5].

3.1. Virtual file system

For our implementation, we utilized one of SQLite3's pre-built in-memory file system called memvfs. In memvfs, each open file has the type of MMemFile (1) where the relevant fields are

Algorithm 1 MMemFile

```
1: struct MMemFile {
2: sqlite3_file base;
3: MMemStore *pStore;
4: int eLock;
5: };
```

In (1), the base stores the methods for performing I/O operations (in-memory) for a file. pStore is a pointer to the underlying storage struct of type MMemStore (2). Finally, eLock is the most recent lock against the file by a connection.

MMemStore (2) stores the storage struct of the file, and contains metadata to mediate access to the file across connections. That is, each connection gets a unique file (of type MMemFile), which contains a pointer to MMemStore which can be shared if the connections open the same file by name (stored in zFName of MMemStore). (2) has the relevant fields, with our additions.

Algorithm 2 MMemStore

```
1: struct MMemStore {
2: unsigned char *aData;
3: wait_queue_head_t write_q;
4: wait_queue_head_t read_q;
5: int num_readers;
6: int current_max_lock;
7: char *zFName;
8: // other fields
9: };
```

aData contains a pointer to dynamically allocated memory which stores the contents of the file. num_readers stores the number of active readers. It should be noted that a connection performing a write will also be included as a reader. current_max_lock is the maximum of all the locks across all open connections and was added as part of making the original implementation of memvfs more concurrent. Specifically, the original implementation followed a strict reader/writer mechanism and does not have the pending lock, and thus can cause writer starvation.

Additionally, wait_queue_head_t write_q and wait_queue_head_t read_q are write and read waitqueues added to block connections if the current lock on the file is incompatible with the requested lock. The functions responsible for locking and unlocking a file are MMemLock and MMemUnlock respectively. SQLite upgrades an existing lock (say, SHARED to RESERVED) by calling the lock function. Similarly, SQLite downgrades an existing lock (say, EXCLUSIVE to NONE) by calling the unlock function. The pseudocode for modified lock and unlock operations are provided in (3) and (4). and (4)

Algorithm 3 MMemLock Algorithm

```
procedure MMemLock(int eLock)
    currentLock = current_max_lock from MMemStore
    if eLock = SHARED then
        if currentLock ≤ RESERVED then
            increment readers
            goto exit
        wait on read waitqueue
    else if eLock = RESERVED then
        goto exit
    else if eLock = EXCLUSIVE then
        if readers = 1 then
            goto exit
        currentLock ← PENDING
        wait on write waitqueue
    exit:
    currentLock ← eLock
    return 0
```

Algorithm 4 MMemUnLock Algorithm

```
procedure MMemUnlock(int eLock)
    currentLock = current_max_lock from MMemStore
    woke_up_writer ← False
    if eLock = SHARED then
        This is currently treated as a no-op
    else if eLock = NONE then
        decrement readers
        if readers = 0 then
            currentLock ← NONE
        else if readers = 1
            & currentLock = PENDING then
            woke_up_writer ← True
            wake up a process on the write waitqueue
    if !woke_up_writer then
        wake up all processes on the read waitqueue
    return 0
```

Here, the MMemLock algorithm assumes that there can only be one request for RESERVED lock. That is, no process will request a RESERVED lock if another process already requested it. To implement this, writes are protected itself by a mutex. Thus, connections that know that there will be updates inside of a transaction, initially acquire this global write mutex and then begin performing queries. The write waitqueue is solely to put this connection to sleep if there are current readers, during the time of commit, and not before that. That is, if there are readers present when the transaction wants to commit, the transaction will be put to sleep on the write waitqueue, while a pending lock will be acquired to prevent new readers (which will sleep on the

read waitqueue) till the write transaction finishes its commit (or rollbacks).
It should be noted that purely read-only connections, do not have to explicitly handle any locking, as it will be handled by the MMemLock and MMemUnlock functions. Additionally, locking the global write mutex does not block new readers (with SHARED lock), until the writer commits. This is because SHARED locks (from readers) can coexist with RESERVED locks (from the writer). When the writer commits, it requests an EXCLUSIVE lock, which will then block readers. Thus, this implementation is more concurrent than the default memvfs locking paradigm, in which a writer will block any reader, regardless of whether the writer holds an EXCLUSIVE lock or a RESERVED lock.
Further, this implementation does not cause deadlocks, since there can only be one write-intent transaction active at a time, guaranteed by acquiring the global write mutex.

3.2. Pipe implementation

Pipes are a form of interprocess unidirectional communication, and the Linux kernel implements pipes using a virtual file system in fs/pipe.c. We implemented IPC, analogous to pipes, backed by the SQLite database. Since implementing pipes was a database problem, we first describe the *schema* (alternatively, set of tables), and detail the problems solved by the chosen schema. Next, we detail the 3 system calls added to implement the IPC, and the SQL queries.

3.2.1. Schema. In databases, schema refers to the set of tables, along with the column names, types, and foreign keys which completely describe the logical structure of the data being stored.

Our schema consists of three tables: **pid_store**, **pid_pipe** and **pipe_data** which are described in detail below.

pid_store: The table consists of 2 columns, **pid** (type **int**) and **handler_id** (type **int**). When a process forks, the new pid, and the handler's pid is inserted into this table. Here, the handler pid is the *usable* index into other system tables. For example, threads created in user space share the parent's file descriptors. In this case, the parent's pid must be used to index tables related to files. Consequently, for fork calls that result in copying of the parent's file descriptors, the handler_id will be the same as the newly created process' pid as the new process does not share the same file descriptor as the parent. Since a process can create multiple threads, the relationship between hander_id and pid in the pid_store table is modeled as many-to-one.

pid_pipe: The table consists of 3 columns, **pid** (type **int**), **fd** (type **int**), and **data_id** (type **int**). The table stores the foreign key to **pid_store)** table, file descriptor, and the foreign key to the **pipe_data** table. The table gets populated when a process calls the new pipe system call, as described in the following subsections. Since a process can have multiple files open, the relationship to the pid_store table is modeled as many-to-one. Additionally, since the file descriptors get copied during the creation of a new process (not thread), multiple pid_pipe records can have the same data buffer, and hence, the relationship to pipe_data table is modeled as many-to-one.

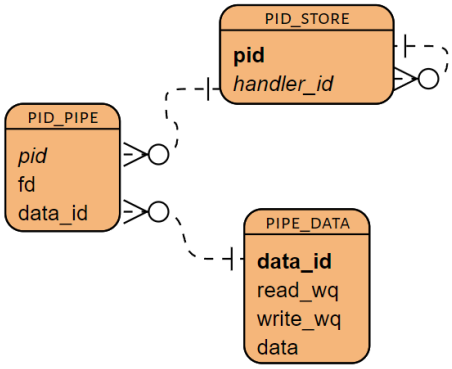

Figure 1. Schema for pipes

pipe_data: The table consists of 4 columns: **data_id** (type **int**), **read_wq** (type **int64**), **write_wq** (type **int64**), and **data** (type **BLOB**). The read_wq and write_wq are pointers to waitqueues for implementing the blocking behavior of pipe system calls. The data column stores the data of pipes as BLOB (binary data).

Figure 1 shows the schema as an E/R diagram, with the tables, and their relationships.

During the boot-up, the modified in-memory file system gets mounted, and the three tables with no records get created.

3.2.2. Pipe Initialization. Traditionally, Linux has the pipe() system call to initialize pipes. In our implementation, we make a new system call to initialize pipes. The signature of our pipe system call is (1), which is similar to the traditional pipe system call.

$$\textbf{long}\ \text{sql\_pipe}(\textbf{int}\ \text{fd}[2]) \quad (1)$$

During the above system call, the kernel first creates a record in the pipe_data table, with pointers to the initialized read and write waitqueues. Next, two unused file descriptors are allocated, for the read and write end of the pipe respectively. Finally, two records are created in the pid_pipe table, one with the read fd and the other with the write fd. Both of these records have the same data_id, pointing to the newly created record in the pipe_data table. It should be noted that the handler_id from the pid_store table is used as the pid in the pid_pipe table, and hence, the pid_store table is queried for the handler_id before the insertion of records in the pid_pipe table.

3.2.3. Pipe Reads. Since the traditional pipes are implemented as a virtual file system, the read() system call can be used to read data from the read end of the pipe. While it is theoretically possible to modify the existing read() system call to instead use the SQLite-backed implementation, we

add a new system call to read data from the pipe. The signature of our pipe read call is (2), which is the same as the traditional read() system call's type signature.

long sql_pipe_read(**int** fd, **char*** buff, **size_t** size) (2)

To perform read from the pipe file descriptor of the calling process, a single query is made which retrieves the read_q, write_q, sliced data (sliced with the given length), and the initial length of the data from the pipe_data table. It should be noted that this query operates on the join of pipe_data table with pid_pipe and pid_store table, to look up the data_id. If the pipe is empty (the initial length is 0), then the corresponding read waitqueue is obtained by dereferencing the read_q pointer, and the process is put on an interruptible sleep. After waking up from sleep (by a writer), the query is re-run, and if the pipe, after reading, is not empty, then the next process on the read waitqueue is woken up. This is done to handle the case where multiple processes sleep on the waitqueue because the pipe was initially empty
Hence, the above pipe read implementation is similar to Linux's implementation, in that the readers block until data is written in the pipe. The algorithm is described in detail in (5). It is worth noting that even though condition (**length** $= 0$) is checked, then the process prepares to sleep, it will not create the problem of "lost wakeup". This is because any transaction needs to acquire the write mutex to change the **length**, which we acquire in the very beginning. Further, the implementation required handling SQLite errors, which has been omitted in the (5) for brevity.

3.2.4. Pipe Writes. Similar to the read system call, the write() system call can be used to write data to the pipe. In our implementation, we add another system call to perform write operations on the SQLite-backed pipes. The type signature of our write system call is (3)

long sql_pipe_write(**int** fd, **char*** buff, **size_t** size) (3)

To perform writes, the kernel first fetches the existing data, and read_q in the pipe_data, and then concatenates the existing data with the input data. Finally, a process on the read_q is woken up. Note that even if multiple readers can be blocked on the same pipe, we still wake up just one reader, which subsequently wakes up the next reader if the pipe is still not empty, after reading. The algorithm is detailed in (6). Similar to pipe reads, the implementation required gracefully returning SQLite errors, which have been omitted for brevity.

4. Evaluation

To test for correctness and execution time differences, both authors compiled the kernel on our personal computers, and ran a small selection of programs.

The initial tests involved running toy examples of pipes between parent and child processes after a fork. Next, lmbench's pipe system call benchmark was run, to test the stability of the implementation under established benchmarks on two different systems. The benchmark results, along with the system specifications, are presented in Table 1. The database-backed implementation does suffer from a major slowdown, which is expected as the pipe logic is implemented at a higher level of abstraction.

Algorithm 5 Pipe Read Steps

```
current is the active process (caller of system call)
procedure SQL_PIPE_READ(int fd, char * buff, size_t size)
    Lock write mutex
    wake_up_next_reader ← False
    while True do
        BEGIN TRANSACTION
        SELECT
            read_wq,                          ▷ Stored as read_wq
            write_wq,
            substr(data, 1, size),            ▷ Stored as read_data
            length(data),                     ▷ Stored as length
            pipe_data.data_id                 ▷ Stored as data_id
        from pid_pipe join pipe_data
            on pipe_data.data_id = pid_pipe.data_id
            join pid_store
            on pid_store.handler_id = pid_pipe.pid
        where fd = fd and pid_store.pid = current⟶ pid
        if length = 0 then
            wake_up_next_reader ← True
            COMMIT                ▷ This is an empty commit
            Unlock write mutex
            read_waitqueue = *(read_wq)
            sleep on read_waitqueue
            Lock write mutex
            continue
    bytes_read = min(size, length)
    UPDATE
      pipe_data
      SET data = SUBSTR(data, bytes_read+1)
    where data_id = data_id
    copy read_data to user space
    if wake_up_next_reader then
        sync wake up a process on read_waitqueue
    COMMIT                        ▷ This will commit changes
    Unlock write mutex
    return bytes_read or any SQLite error
```

TABLE 1. PIPE BANDWIDTH IN MB/SEC

CPU	AMD Ryzen 5 5600	i5-13600KF
Memory	16GB	32GB
Pipe Bandwidth (SQLiteOS)	124.8	102.18
Pipe Bandwidth	1852.33	3271.53

4.1. Stability

As LMbench did not cause any crashes or error, nor did our other tests involving pipes and forks, it is clear that our patching does not add any real system instability or

Algorithm 6 Pipe Write Steps

current is the active process (caller of system call)
procedure SQL_PIPE_WRITE(**int** ***fd***, **char** * ***buff***, **size_t** ***size***)
    Lock write mutex
    BEGIN TRANSACTION
    SELECT
      data, ▷ Stored as **existing_data**
      length(data), ▷ Stored as **length**
      pipe_data.data_id, ▷ Stored as **data_id**
      read_wq ▷ Stored as **read_wq**
    from pid_pipe join pipe_data
      on pipe_data.data_id = pid_pipe.data_id
      join pid_store
      on pid_store.handler_id = pid_pipe.pid
    where fd = ***fd*** and pid_store.pid = ***current***⟶ ***pid***;
    new_data ← concat(**existing_data**, ***buff***)
    UPDATE
      pipe_data
      SET data=new_data
    where data_id = **data_id**
    COMMIT
    Unlock write mutex
    read_waitqueue = *(**read_wq**)
    wake up a single process on read_waitqueue
    return bytes written or any SQLite error

error. Particularly, LMbench tests pipes and forks under high contention, so we would expect to see such failures here in particular. We conclude that so far, our kernel patch is stable. It should be noted that the kernel image size increased by 496,128 bytes, and the final image size was 14,874,816 bytes.

5. Future Directions

While we generally met our targets, we immediately identified several core areas for potential improvement. Mostly, these have to do with the slowdowns we observed.

The current handling for conflicting transactions is sequential via a queue, with mutex passing permitting write control. A future point of investigation may be optimizing these queues with a weighting system, and finding alternative means of mutex passing, if mutexes are indeed necessary at all.

The Virtual File System on which the database table is held is presently not optimized. We are currently not confident that memory reallocation is as efficient as it could be in cases where new memory must be allocated as the table expands. Ensuring that previously used memory is appropriately freed back to user space would be a priority.

As we suspect SQLite3 locking the entire table during a write operation contributes heavily to observed slowdowns, we believe allowing row-level locks would alleviate this. There are two means to achieve this - there is at least one extant fork of SQLite3 which allows row level locking. Alternatively, we could patch in this functionality into SQLite3 ourselves. Regardless, we suspect this fix would also implicitly address the prior concerns as well.

6. Conclusion

While there are refinements to be made, we conclude that a database-backed operating system shows potential. Additionally, from our experience of implementing pipes, it was observed that the ACID guarantees of the DBMS promote a simplified implementation since race conditions are automatically handled by the database, as long as the underlying locking mechanism is correctly implemented (which took the form of MMemLock and MMemUnlock in our implementation). Further, we show that integrating pre-existing functionality (such as waitqueues) is possible by storing pointers to the relevant data structures.

Altogether, surgical and concise modifications of the SQLite3 and the stock WSL2 kernel allowed for a radical overhaul of the Unix/Linux paradigm. These changes, estimated at about 1,100 lines of code total, transformed the 'everything is a file' kernel to one that stably holds process IDs and pipe file descriptors in a database table.

References

[1] Microsoft *WSL2-Linux-Kernel*, https://github.com/microsoft/WSL2-Linux-Kernel. Accessed at 5/12/2024

[2] L. McVoy and C. Staelin, *LMbench - Tools for Performance Analysis*, https://lmbench.sourceforge.net/. Accessed at 5/12/2024

[3] A. Skiadopoulos et al, *DBOS: A DBMS-oriented Operating System*, PVLDB, vol. 15, no 1, 2022. : 21-30. DOI:10.14778/3485450.3485454

[4] Q. Li et al. *A Progress Report on DBOS: A Database-oriented Operating System*, Presented at CIDR, Chaminade, CA, USA, Jan 9-12, 2022. Paper 26

[5] SQLite Consortium, *SQLite*, https://www.sqlite.org/. Accessed at 5/12/2024

[6] SQLite Consortium, *Well-Known Users of SQLite*, https://www.sqlite.org/famous.html. Accessed at 5/12/2024

[7] SQLite Consortium, *The SQLite OS Interface or "VFS"*, https://www.sqlite.org/vfs.html. Accessed at 5/12/2024

[8] SQLite Consortium, *File Locking And Concurrency In SQLite Version 3*, https://www.sqlite.org/lockingv3.html. Accessed at 5/12/2024