Guidedog Discussions GDS 0019

File locks

STATE
committed
TYPE
Standards Track
AUTHORS
Vikrant Rathore
CREATED
2026-11-23
UPDATED
2026-11-23
DISCUSSION
No external discussion URL assigned
LABELS
library

Record status

The catalog defines the review process and the meaning of each state. This numbered record forms part of the catalog.

Contents

1. Abstract 1
2. Motivation and scope 1
3. Specification 2
4. Rationale and alternatives 7
5. Security and operational considerations 9
6. Backwards compatibility 9
7. Acceptance criteria and implementation status 9
8. Open questions 10
9. Review history 10
10. References 10

1. Abstract

Guidedog serializes the commands that change a small piece of state with an advisory lock on a lock file. We specify one lock for all of them, `gd_plugins::files::lock_exclusive`, and one rule: a holder releases the lock with an explicit unlock, and only then closes the file. A command is then never told that a lock is held when no command holds it. Before, a holder released by closing the file, and a child process that another thread had just created kept the lock alive for an instant. We model the lock, the takers, the children, crashes and the update of the state file as a transition system. We prove in Lean that the algorithm keeps an invariant, and from it mutual exclusion, the absence of a false refusal, recovery from a crash, and that readers see whole files and writers lose no update. We also prove that the algorithm we replace reaches a false refusal.

2. Motivation and scope

The defect. `guidedog plugin register` takes the lock file `plugins.toml.lock` with `std::fs::File::try_lock` and reports `plugins.busy` when the lock is held. The test `an_installed_publisher_publishes_through_the_protocol` registers a program twice in a row and expects the second call to say `plugins.registered`. On the macOS

runner of the verification workflow the second call said `plugins.busy`, and the same happened on Linux in a fraction of the runs. Nothing else was changing that registry.

The cause. On Unix, `try_lock` is `flock`. A `flock` lock belongs to the open file description, not to the descriptor and not to the process [flock2]. Every descriptor that was copied from the one we opened shares the lock, and the system releases the lock by itself only when the last of them closes. A child process receives copies of all descriptors of its parent when it is created [fork2]. The descriptors Rust opens are close-on-exec, so the child's copies close when it starts its program [execve2], but they exist until then. The tests of one binary run as threads of one process, and another test starts plugin programs. So a child created while the first `register` had the lock file open held a copy of that descriptor. The first `register` closed its own descriptor, the copy kept the lock, and the second `register` was refused. Go's issue 22315 describes the same window for a file that is being written and then executed [go22315].

The evidence. We ran the test binary of `crates/gd-plugins/tests/publishers.rs` 1000 times before and 1000 times after the change, four runs at a time, on Linux 7.0 (x86-64, 32 processors). Before, built from the commit of 1.0.0-beta.14, 34 of 1000 runs failed, each with `plugins.busy`. After, none of 1000 runs failed. The binary after the change has one more test, which starts programs on two threads while two other threads register and unregister 100 times each. A test that needs no timing shows the defect directly: it duplicates the lock's descriptor, which shares the open file as a child's copy does, releases the lock, and takes it again. A second test closes a lock file without unlocking while a copy lives, to show the leak. On 2026-11-23 that test itself met the leak on macOS (Apple silicon) in the verification workflow: a child that another test of the same binary had started held one more copy when the test expected the lock to be free. This is one more observation of the defect, on a second platform; the test now waits for the release (commit `827e39c`), and the library did not change.

What was there. Five places locked this way, and the example publisher a sixth. Two of them had met the defect before and each had its own remedy: `gd-pages` unlocked before closing, and `gd-discussion` asked again for 200 ms. The registry had a retry of 500 ms that was never committed. A retry makes the false refusal rare and a true refusal slow; it does not remove the cause.

Scope. This record covers the lock of the plugin registry, of a publisher's state in `gd-pages`, of a discussion catalog, of a site's sources and of a whole-site build in `gd-ssg`, and the lock the example publisher takes. It covers Linux, macOS and Windows, the systems Guidedog is released for, and FreeBSD, whose `flock` has the same rules. Locks inside one process (`std::sync::Mutex`) and the journal of a discussion catalog (GDS 0007) are out of scope.

3. Specification

The interface. `gd_plugins::files` gains four items.

```
pub struct FileLock { /* the open lock file */ }
pub enum LockError { Held, Io(std::io::Error) }
pub fn lock_exclusive(path: &Path) -> Result<FileLock, LockError>;
pub fn lock_exclusive_private(path: &Path) -> Result<FileLock, LockError>;
```

`lock_exclusive` opens the lock file `path`, creating it if it is missing, and asks for the exclusive lock without waiting. It returns the lock, or `LockError::Held` when another open file holds it, or `LockError::Io` when the file cannot be opened or locked. `lock_exclusive_private` also makes a file it creates readable and writable only by its user. Dropping the `FileLock` unlocks and then closes the file. The lock file stays: removing it would let two takers lock different files at one path. Each caller turns `Held` into its own problem, so `plugins.busy`, `pages.busy`, `PREFIX.busy`, `ssg.conflict`, `ssg.site` and `folder.busy` keep their codes, messages and exit statuses. No caller waits or asks again.

```
let _lock = match lock_exclusive_private(&lock_path) {
    Ok(lock) => lock,
    Err(LockError::Held) => return Err(busy_problem(path)),
    Err(LockError::Io(e)) => return Err(io_problem(&lock_path, &e)),
};
let mut registry = read_machine(path)?;      // read, under the lock
registry.entries.push(entry.clone());
```

```
write_machine(path, &registry.entries)?;     // draft, sync, rename
// the lock is released here: unlock, then close
```

The model. A *taker* is one call of `lock_exclusive` together with the open file that call creates. We name takers by natural numbers; a second call is another taker. A state σ has seven parts.

$$
\begin{aligned}
\mathit{phase} &: \mathbb{N} \to \{\text{idle}, \text{opened}, \text{held}, \text{released}, \text{done}, \text{dead}\} \\
\mathit{locked} &: \mathbb{N} \to \{0, 1\} \\
\mathit{copies} &: \mathbb{N} \to \mathbb{N} \\
\mathit{seen} &: \mathbb{N} \to \mathbb{N} \cup \{\bot\} \\
\mathit{stage} &: \mathbb{N} \to \{\text{empty}, \text{part}, \text{whole}\} \\
\mathit{file} &\in \mathbb{N} \times \{0, 1\} \\
\mathit{next} &\in (\mathbb{N} \times \{0, 1\}) \cup \{\bot\}
\end{aligned}
$$

$\mathit{phase}\ (a)$ says where the taker a is in its call. It is held while a is in its critical section, and dead when its process ended without releasing. $\mathit{locked}\ (a) = 1$ says that the system holds the lock for the open file of a. This is the system's state, not the taker's belief. $\mathit{copies}\ (a)$ counts the copies of a's descriptor in children that have not yet started their programs. file is the state file as a reader sees it, a value and a flag that says whether every byte is there. next is the file a writer prepares beside it (`plugins.toml.next`), $\mathit{seen}\ (a)$ is what the holder a read, and $\mathit{stage}\ (a)$ says how far it wrote. In the initial state every taker is idle, nothing is locked or copied, and $\mathit{file} = (0, 1)$.

We write $\mathit{open}\ (a)$ for $\mathit{phase}\ (a) \in \{\text{opened}, \text{held}, \text{released}\}$: the taker has its own descriptor open. Two functions state the system's rule for a lock when a descriptor closes. When a's own descriptor closes, the lock stays only if a copy is left. When a copy closes, the lock goes if that was the last descriptor.

$$
\begin{aligned}
\mathit{afterClose}\ (a) &= \mathit{locked}\ (a) \wedge \mathit{copies}\ (a) \neq 0 \\
\mathit{afterExec}\ (a) &= \mathit{locked}\ (a) \wedge \neg(\mathit{copies}\ (a) = 1 \wedge \mathit{phase}\ (a) \in \{\text{done}, \text{dead}\})
\end{aligned}
$$

The algorithm as transitions. Each row is one atomic step of one taker a, or of a child that copies its descriptor. A step may happen when its guard holds, and changes only what its row names. The last column names the code that takes the step.

Table 36 The transitions

Step	Guard	Effect	Code
`openFile`	$\mathit{phase}\ (a) = \text{idle}$	$\mathit{phase}\ (a) := \text{opened}$	`OpenOptions::open` in `files/lock.rs`
`acquire`	$\mathit{phase}\ (a) = \text{opened}$ and $\forall b.\ \mathit{locked}\ (b) = 0$	$\mathit{phase}\ (a) := \text{held}$, $\mathit{locked}\ (a) := 1$, $\mathit{seen}\ (a) := \bot$, $\mathit{stage}\ (a) := \text{empty}$	`try_lock` returns `Ok`
`refuse`	$\mathit{phase}\ (a) = \text{opened}$ and $\exists b.\ \mathit{locked}\ (b) = 1$	$\mathit{phase}\ (a) := \text{done}$, $\mathit{locked}\ (a) := \mathit{afterClose}\ (a)$	`try_lock` returns `WouldBlock`; the caller reports its busy problem
`unlock`	$\mathit{phase}\ (a) = \text{held}$	$\mathit{phase}\ (a) := \text{released}$, $\mathit{locked}\ (a) := 0$	`File::unlock` in `FileLock::drop`
`close`	$\mathit{phase}\ (a) = \text{released}$	$\mathit{phase}\ (a) := \text{done}$, $\mathit{locked}\ (a) := \mathit{afterClose}\ (a)$	the file field drops after `FileLock::drop`
`fork`	$\mathit{open}\ (a)$	$\mathit{copies}\ (a) := \mathit{copies}\ (a) + 1$	any `Command::spawn` in the taker's process
`exec`	$\mathit{copies}\ (a) > 0$	$\mathit{copies}\ (a) := \mathit{copies}\ (a) - 1$, $\mathit{locked}\ (a) := \mathit{afterExec}\ (a)$	the child starts its program, or exits
`crash`	$\mathit{open}\ (a)$	$\mathit{phase}\ (a) := \text{dead}$, $\mathit{locked}\ (a) := \mathit{afterClose}\ (a)$	the process ends: exit, abort or kill
`read`	$\mathit{phase}\ (a) = \text{held}$	$\mathit{seen}\ (a) := \mathit{file}_1$	`read_machine` in `register` and `unregister`

Step	Guard	Effect	Code
`draft`	$phase\ (a) = \text{held}$ and $seen\ (a) \neq \bot$	$next := (v, 0)$, $stage\ (a) := \text{part}$	`write_atomically`: remove and create the file beside
`sync`	$phase\ (a) = \text{held}$, $stage\ (a) = \text{part}$ and $next = (v, 0)$	$next := (v, 1)$, $stage\ (a) := \text{whole}$	`write_atomically`: `write_all` and `sync_all`
`rename`	$phase\ (a) = \text{held}$, $stage\ (a) = \text{whole}$ and $next = w$	$file := w$, $next := \bot$, $stage\ (a) := \text{empty}$, $seen\ (a) := \bot$	`write_atomically`: `fs::rename`

The algorithm we replace has one more step, `closeHeld`: a holder goes from held to done with $locked\ (a) := afterClose\ (a)$, without unlocking. We write $Reach$ for the states the algorithm reaches from the initial state, and $Reach_0$ for those the replaced algorithm reaches.

The model does not say which takers share a process, and it does not need to. A child creation that copies several descriptors is several `fork` steps, one after another, and every state between them is a state the model already allows. The end of a process with several takers is a `crash` step of each. So every run of the real system is a run of the model, and what holds in all runs of the model holds in the real ones.

The invariant. For all takers a, b and all values u:

(I1) $locked\ (a) \wedge locked\ (b) \Rightarrow a = b$

(I2) $phase\ (a) = \text{held} \Rightarrow locked\ (a)$

(I3) $locked\ (a) \Rightarrow phase\ (a) = \text{held} \vee (phase\ (a) = \text{dead} \wedge copies\ (a) > 0)$

(I4) $file_2 = 1$

(I5) $phase\ (a) = \text{held} \wedge stage\ (a) = \text{whole} \Rightarrow \exists v.\ \ next = (v, 1)$

(I6) $phase\ (a) = \text{held} \wedge seen\ (a) = u \Rightarrow file_1 = u$

(I7) $phase\ (a) = \text{held} \wedge stage\ (a) \neq \text{empty} \Rightarrow seen\ (a) \neq \bot$

I1 is the system's promise that one open file has the lock at a time. I2 and I3 tie the system's lock to the taker's critical section in both directions; I3 is the clause the replaced algorithm breaks. I4 to I7 are about the state file.

Lemma 1 (the invariant is inductive). I1 to I7 hold in the initial state, and every step of the algorithm keeps them. So they hold in every state of $Reach$. In Lean: `inv_init`, `inv_step` (one lemma per step, `inv_openFile` to `inv_rename`) and `inv_reachable`.

Proof. In the initial state nothing is locked and no taker is held, so I1 to I3 and I5 to I7 hold with false premises, and $file = (0, 1)$ gives I4. For the steps we take each clause in turn and name the steps that can make it false.

I1. Only `acquire` sets a lock. Its guard says that no lock is set, so afterwards a alone is locked.

I2. `acquire` makes a held and locked together. A lock of a is cleared by `unlock` and `crash`, which take a out of held; by `refuse` and `close`, whose taker is not held; and by `exec`, which clears it only when a is done or dead.

I3. `acquire` gives the first alternative. `unlock` leaves held and clears the lock. `crash` leaves held for dead, and $afterClose$ keeps the lock only if $copies\ (a) \neq 0$, which is the second alternative. `exec` lowers $copies\ (a)$: for a dead taker with one copy, $afterExec$ clears the lock; with more copies the count stays positive; for a held taker the first alternative stands. `fork` only raises the count, and a dead taker has no descriptor to copy. `refuse` and `close` start from opened and released, where I3 says the taker is not locked, and $afterClose$ sets no lock.

I4. Only `rename` changes $file$. Its taker is held with $stage\ (a) = \text{whole}$, so by I5 $next = (v, 1)$, and that is the new $file$.

I5. `sync` by a sets $next = (v, 1)$ and $stage\ (a) = \text{whole}$ together. $next$ is changed only by `draft`, `sync` and `rename`, each by a held taker b. If a is held too, I2 and I1 give $b = a$, and `draft` and `rename` by a leave $stage\ (a) \neq \text{whole}$. `acquire` empties the stage, so a new holder starts with a false premise.

I6. `read` sets $seen\ (a) = file_1$. $file$ is changed only by `rename` by a held b, again $b = a$, and that step clears $seen\ (a)$. `acquire` clears it for a new holder.

I7. `draft` requires $seen\ (a) \neq \bot$. $seen\ (a)$ is cleared only by `acquire` and `rename`, which also empty the stage. □

Theorem 1 (mutual exclusion).

$$\sigma \in Reach \land phase\ (a) = \text{held} \land phase\ (b) = \text{held} \Rightarrow a = b$$

Proof. By I2 both are locked, and by I1 they are one taker. □ In Lean: `mutual_exclusion`. This holds for takers in one process and in several, because separate opens conflict (assumption A3 below).

Theorem 2 (no false refusal). A taker is refused when the guard of `refuse` holds.

$$\sigma \in Reach \land (\exists b.\ \ locked\ (b)) \Rightarrow \exists b.\ \ phase\ (b) = \text{held} \lor (phase\ (b) = \text{dead} \land copies\ (b) > 0)$$

Proof. I3 for the locked b. □ In Lean: `no_false_busy`.

Corollary 2a. While no taker has died, a refusal means that a taker is in its critical section.

$$\sigma \in Reach \land (\forall b.\ \ phase\ (b) \neq \text{dead}) \land (\exists b.\ \ locked\ (b)) \Rightarrow \exists b.\ \ phase\ (b) = \text{held}$$

In Lean: `no_false_busy_alive`. This is the property the defect broke. A copy of the descriptor in a child does not appear in it: copies of a live taker cannot cause a refusal, whenever they close.

Corollary 2b. A lock that is held is refused: $\sigma \in Reach \land phase\ (b) = \text{held} \Rightarrow \neg\forall c.\ \ locked\ (c) = 0$, so `acquire` cannot happen. In Lean: `held_is_refused`.

Theorem 3 (readers see whole files). $\sigma \in Reach \Rightarrow file_2 = 1$. This is I4. In Lean: `reader_sees_whole`. A reader takes no lock.

Theorem 4 (no lost update). When a holder is about to rename, it has read the state file, the state file still holds what it read, and the file beside is whole.

$$\sigma \in Reach \land phase\ (a) = \text{held} \land stage\ (a) = \text{whole} \Rightarrow \exists u, v.\ \ seen\ (a) = u \land file_1 = u \land next = (v, 1)$$

Proof. I7 gives u, I6 gives $file_1 = u$, and I5 gives v. □ In Lean: `no_lost_update`. So the values of the state file form one chain: each rename replaces exactly the value its writer read. In the words of Herlihy and Wing [herlihy-wing], the read-modify-write behaves as one atomic operation that takes effect at the rename.

Theorem 5 (crash recovery). A dead taker whose descriptor no child copies holds no lock.

$$\sigma \in Reach \land phase\ (b) = \text{dead} \land copies\ (b) = 0 \Rightarrow locked\ (b) = 0$$

Proof. By I3 a locked taker is held, or dead with a copy. □ In Lean: `dead_holds_nothing`.

Theorem 6 (the lock is free). When no taker is held and no dead taker's descriptor is copied, nothing is locked.

$$\sigma \in Reach \land (\forall b.\ \ phase\ (b) \neq \text{held}) \land (\forall b.\ \ phase\ (b) = \text{dead} \Rightarrow copies\ (b) = 0) \Rightarrow \forall b.\ \ locked\ (b) = 0$$

Proof. I3 again. □ In Lean: `lock_is_free`. *Corollary 6a*: in such a state a taker that opened the lock file takes the lock in one step (`taker_succeeds`).

Progress. The safety theorems say what never happens. For progress we must assume something about the children.

Assumption F. Every child that holds a copy of a descriptor eventually starts its program or exits: for each copy, its `exec` step eventually happens.

Lemma 7. No step revives a dead taker or adds a copy of its descriptor: if $phase\ (b) = \text{dead}$ in σ and $\sigma \to \sigma'$, then $phase'(b) = \text{dead}$ and $copies'(b) \leq copies\ (b)$. Proof. `fork` and `crash` need an open descriptor, and every other step that changes a phase needs a phase other than dead. □ In Lean: `dead_copies_never_grow`; it holds of the replaced algorithm too.

Theorem 8 (bounded recovery). From a reachable state where b is dead, the $copies\ (b)$ `exec` steps of the children that copy its descriptor lead to a reachable state where b holds no lock. Proof. By induction on $copies\ (b)$. With no copy this is Theorem 5. Otherwise `exec` may happen, keeps b dead and lowers the count by one. □ In Lean: `crash_recovers`.

Together these give the progress property. By Lemma 7 the number of copies of a dead holder's descriptor never grows, by assumption F each of them goes, and by Theorem 8 the lock is released when the last one does. Then, by Corollary 6a, a taker gets the lock unless another taker got it first. We prove this bounded form and the lemma in Lean. We do not state F or the word "eventually" in Lean, and we prove no fairness between takers: `lock_exclusive` never waits, so a taker that is refused may be refused again by another holder. A live holder may hold as long as it works; nothing here bounds a critical section.

Theorem 9 (the replaced algorithm is refuted).

$$\exists \sigma \in Reach_0.\ (\forall b.\ \ phase\ (b) \neq \text{held}) \wedge (\forall b.\ \ phase\ (b) \neq \text{dead}) \wedge \exists b.\ \ locked\ (b)$$

Proof. Taker 0 does `openFile` and `acquire`, a child does `fork` of its descriptor, and taker 0 does `closeHeld`. Now $copies\ (0) = 1$, so $afterClose$ keeps the lock, and taker 0 is done. □ In Lean: `close_alone_refuses_falsely`. This is the failing test, step for step.

What ties the model to the system. The proofs are about the model. The model is right about the real system as far as these assumptions hold.

Table 37 Assumptions about the system

No.	Assumption	Source
A1	A `flock` lock belongs to the open file. Descriptors copied by `dup` or `fork` share it.	[flock2]; the macOS and FreeBSD pages say "Locks are on files, not file descriptors" [flock-bsd]
A2	The lock is released by an unlock through any of these descriptors, or when all of them have closed, and by nothing else.	[flock2]; Rust's `File::lock` documents the same for both systems [rust-file]
A3	Two opens of one file conflict, in one process too.	[flock2]; for Windows [lockfileex]
A4	A child gets copies of its parent's descriptors when it is created. A close-on-exec descriptor closes when the child starts its program. Rust opens every file close-on-exec and duplicates with close-on-exec.	[fork2], [execve2], [open2]; Rust's standard library source [rust-file]
A5	When a process ends, the system closes its descriptors.	POSIX `_exit`; for Windows [lockfileex]
A6	`rename` replaces the destination in one step: a reader that opens the name gets the old file or the new one.	POSIX `rename` [rename]
A7	A reader that has opened the state file reads that file to its end, whatever is renamed over its name.	POSIX `rename` and `open`
A8	Every program that changes the state takes the lock. The lock is advisory.	This record

What the model leaves out. It has no power loss: `sync_all` before the rename is there for it, and we do not prove it. It has no failure of `unlock`; if it fails, the close that follows releases the lock as before, with the old window. It has one state file per lock; the several files of a discussion update are made recoverable by their journal (GDS 0007), not by this proof. It assumes the lock file is never removed or replaced. It has no network file system: Linux emulates `flock` over NFS with process-owned locks [flock2], where A1 and A3 fail.

The six users.

Table 38 Where the lock is taken

Place	Lock file	Held for	When held
`gd_plugins::register`, `unregister`	`plugins.toml.lock`, private	read, change, replace the registry	`plugins.busy`
`gd_pages::State::lock`	`<target>.lock` and the settings' locks, private	a publication, which runs `git`; a change of settings	`pages.busy`
`gd_discussion::lock_catalog`	`.PREFIX.lock`	load, plan and apply a change	`PREFIX.busy`
`gd_ssg` source update	`.guidedog-ssg.lock`	compare, write and rename one source	`ssg.conflict`
`gd_ssg::build_site`	`site.lock` in the build folder	a whole-site build, which starts plugin programs	`ssg.site`
the example publisher	`<target>.lock`	a publication session	`folder.busy`

4. Rationale and alternatives

Who owns a lock. The defect is a question of ownership, so we first set down who owns each kind of lock and what a child receives.

Table 39 Lock ownership

Lock	Owner	A child process	Released
`flock` (Linux, macOS, FreeBSD)	the open file	shares it through its copies of the descriptors, until it starts its program	by an unlock through any copy, or when the last copy closes [flock2] [flock-bsd]
POSIX record lock (`fcntl F_SETLK`)	the process	inherits nothing	by an unlock, when the process ends, and when the process closes *any* descriptor of the file [fcntl] [posix-fcntl]
Open file description lock (`F_OFD_SETLK`, Linux 3.15, POSIX.1-2024)	the open file	shares it, as with `flock`	as with `flock` [fcntl] [lwn-ofd]
`LockFileEx` (Windows)	the handle of the locking process	is granted no access, even with an inherited handle; Rust's file handles are not inheritable	by an unlock, or by the system after the handle closes or the process ends [lockfileex]

Process-owned locks do not conflict inside one process, so they cannot serialize threads [fcntl]. SQLite's source calls them "broken by design" and keeps a table of inodes to live with them [sqlite]. Rust's `File::lock` is `flock` on Unix and `LockFileEx` on Windows [rust-file].

The candidates. We compared six designs against eight requirements.

Table 40 Designs compared

Design	No false refusal	Excludes processes and threads	A crash frees the lock	Whole files for readers	Never waits long	Standard library, no unsafe code	Short updates	Long holds that start programs
Close to release (before)	no	yes	yes	yes	yes	yes	yes	yes
Ask again for a while	rare, by timing	yes	yes	yes	a true refusal takes the whole wait	yes	yes	yes
(a) Process-owned locks and a table of held paths	yes	threads only by the table	yes	yes	yes	no: `fcntl` needs foreign calls	yes	yes, but any close of the file drops the lock
(b) No lock: compare and swap on the file system	yes	no lock to hold	a lock file made with `create_new` stays	yes	retries, some writer always proceeds	yes	yes, with a new file layout	no: a publication cannot be retried

Design	No false refusal	Excludes processes and threads	A crash frees the lock	Whole files for readers	Never waits long	Standard library, no unsafe code	Short updates	Long holds that start programs
(c) An ownership record in the lock file	yes, if the taker waits for the leftover copy	yes	needs to know whether a process is alive	yes	a wait for the copy	not for the liveness test	yes	yes
(d) A gate that forbids starting programs while a lock file is open	only for programs we start	yes	yes	yes	yes	yes	yes	no: the holder must start programs
Unlock, then close (chosen)	yes (Theorem 2)	yes (Theorem 1)	yes (Theorems 5, 8)	yes (Theorem 3)	never waits	yes	yes	yes

(a) Process-owned locks. A child inherits none, which removes the leak. But the standard library offers no `fcntl` lock, and Guidedog keeps unsafe code to its foreign-interface modules. The lock is dropped when the process closes any descriptor of the lock file, and it does not exclude threads, so we would need the table SQLite and Go's `fcntl` variant keep [sqlite] [go-lockedfile]. Windows would still need another mechanism.
(b) Optimistic updates. Kung and Robinson split a transaction into a read phase, a validation and a write phase, and keep only validation and write in a critical section [kung-robinson]. On a file system the atomic test is an exclusive create or a hard link, which fails when the name exists [open2]. Git builds its lock from it: it creates `<name>.lock` exclusively, writes the new contents there and renames it over the name [git-lockfile]. That is a lock, and a crashed Git leaves the lock file; Git removes it at exit and on a signal, and after a crash the user removes it. A true compare and swap of the registry needs names that carry a generation, which changes the file users see as `plugins.toml` and adds the cleaning of old generations. In Herlihy's terms [herlihy-waitfree] such an update is lock-free: some writer always completes. It does not fit the long holders at all: a publication pushes to a server and cannot be run again when validation fails. Guidedog keeps the useful part already. A source update of `gd-ssg` and `apply` of `gd-discussion` prepare their change without the lock and validate under it that each file still has the contents the change was made from. So the lock covers validation and write only, as Kung and Robinson prescribe.
(c) An ownership record. The holder would write a mark into the lock file and clear it before closing; a taker that is refused would read the mark and wait when none is set. A crash between the two steps leaves a mark with no holder, and deciding that needs a test whether a process is alive, with the hazard of reused process numbers. The explicit unlock is this design with the system's lock as the mark: the system clears it in one step, for every copy, and clears it by itself when the holder dies.
(d) A gate. Go's `syscall.ForkLock` excludes the creation of a child while a descriptor is being opened [go-forklock]. It exists for systems that cannot open a file close-on-exec in one step, and it does not close the window between creation and program start. A gate around our locks could not hold while a publication starts `git`, and it would not see a program that another library starts.
The choice. We keep the system's lock, because only the system releases a lock when its holder dies without a word, and we make the release an action. Cargo's `FileLock` and Go's `lockedfile` both unlock before they close [cargo-flock] [go-lockedfile]. `gd-pages` already did. We put the algorithm in `gd_plugins::files`, beside `write_atomically`, because every crate that locks already depends on `gd-plugins` and a publisher written against the SDK needs the same lock.

The method. We follow Lamport [lamport77]: a safety property is proved by an invariant that holds initially and that every atomic action keeps, and a liveness property needs its own argument and its own assumption. Owicki and Gries prove the same kind of invariant by showing that no action of one process interferes with the assertions of another [owicki-gries]; our case split on $b = a$ in Lemma 1 is that check.

5. Security and operational considerations

What stays dependent on timing. One case, and it is not a false refusal by Theorem 2. When a holder dies while a child it or a sibling thread just created has not started its program, the system keeps the lock until that child does. A taker that asks in that instant is refused. The same taker an instant earlier would have been refused by the living holder. The window is the time a child needs to start its program, and it needs a crash inside it.

Crashes. A dead holder leaves the lock file, a possibly partial file beside the state file, and the state file untouched (Theorem 3). The next holder removes the file beside before it writes its own.

Windows. A lock belongs to the handle, a child is granted no access through an inherited handle, and Rust's file handles are not inheritable, so the defect did not exist there [lockfileex]. Microsoft's documentation says that the system unlocks after a handle closes or a process ends, that this takes time that "depends upon available system resources", and that a process should unlock explicitly. The explicit unlock follows that advice. After a crash on Windows a taker may be refused for that time.

Network file systems. Over NFS, Linux emulates `flock` with process-owned locks, so two threads of one process do not exclude each other there, and Cargo skips file locks on NFS mounts altogether [cargo-flock]. Guidedog's commands are single processes, so the registry stays safe between commands; we give no promise for threads on NFS.

Trust. The lock is advisory and protects cooperating commands from each other, not from a hostile program of the same user. `gd-ssg` and `gd-discussion` still refuse a lock path that is not a regular file before they open it.

6. Backwards compatibility

Problem codes, messages, exit statuses, lock file names and file formats are unchanged. `gd_discussion::lock_catalog` no longer asks again for 200 ms; it reports a held lock at once, as the other commands always did. `gd_pages::StateLock` and `gd_discussion::Lock` keep their public form. `gd_plugins::files` gains `FileLock`, `LockError`, `lock_exclusive` and `lock_exclusive_private`.

7. Acceptance criteria and implementation status

Implemented in the working tree for 1.0.0-beta.15.

- All six users take the lock through `gd_plugins::files`; no other `try_lock` on a file remains.
- `files::lock::tests::a_copy_of_the_descriptor_does_not_keep_a_released_lock` reproduces the defect without timing and fails when the unlock is removed.
- `closing_alone_leaves_the_lock_with_a_copy` checks assumption A2 on the running system.
- `a_holder_that_exits_without_releasing_leaves_no_lock` runs a process that takes the lock and exits without releasing.
- `the_algorithm_keeps_the_invariant_in_every_interleaving` enumerates the model for two takers with two copies and for three takers with one copy, and checks I1 to I7 and Theorems 2, 4, 5, 6 and 8 in every reachable state. `closing_alone_reaches_a_false_refusal` finds the state of Theorem 9 and checks that the replaced algorithm breaks I3 and nothing else.
- Two stress tests start programs on other threads while threads lock and unlock, and register and unregister.
- `docs/gds/proofs` holds the Lean proofs (`Gds/R0019/FileLock.lean`), for Lean 4.19 with its core library alone. `cargo xtask proofs` builds them, refuses an open proof, and checks that each theorem depends on no axiom beyond `propext`, `Classical.choice` and `Quot.sound`. The theorems here use `propext` and `Quot.sound` only.
- The measured runs are in "Motivation and scope".

We ran everything on Linux x86-64. We did not run the change on macOS or Windows; `gd-plugins` and its tests compile for `aarch64-apple-darwin` and `x86_64-pc-windows-msvc`, and the reasoning for those systems

rests on the documents cited. The verification workflow does not install Lean, so `cargo xtask proofs` is a task we run by hand before a release, not a step of that workflow.

8. Open questions

- Whether the verification workflow should install Lean and run `cargo xtask proofs`.
- Whether to stop locking on network file systems, as Cargo does, or to say only that we give no promise there.

9. Review history

- 2026-11-23: Written with the implementation, after the registry test failed on the macOS runner of the verification workflow. The maintainer asked for a design that removes the race instead of a retry, for a record of its own and for proofs in Lean.
- 2026-11-23: After release 1.0.0-beta.15 was prepared, the test `closing_alone_leaves_the_lock_with_a_copy` failed once on the macOS runner of the verification workflow, because it closes without unlocking and so was open to the leak it shows. The test was corrected to wait for the release; the evidence in *Motivation and scope* records the observation.

10. References

[flock2] `flock(2)`, Linux man-pages 6.19. https://man7.org/linux/man-pages/man2/flock.2.html

[flock-bsd] `flock(2)`, macOS (xnu `bsd/man/man2/flock.2`) and FreeBSD 15.1. https://man.freebsd.org/cgi/man.cgi?query=flock&sektion=2

[fork2] `fork(2)`, Linux man-pages. https://man7.org/linux/man-pages/man2/fork.2.html

[execve2] `execve(2)`, Linux man-pages. https://man7.org/linux/man-pages/man2/execve.2.html

[open2] `open(2)`, Linux man-pages: `O_CLOEXEC` and `O_EXCL`. https://man7.org/linux/man-pages/man2/open.2.html

[fcntl] `fcntl_locking(2)`, Linux man-pages: record locks and open file description locks. https://man7.org/linux/man-pages/man2/fcntl_locking.2.html

[posix-fcntl] The Open Group, POSIX.1-2024, `fcntl()`. https://pubs.opengroup.org/onlinepubs/9799919799/functions/fcntl.html

[rename] The Open Group, POSIX.1-2024, `rename()`. https://pubs.opengroup.org/onlinepubs/9799919799/functions/rename.html

[lwn-ofd] Jeffrey T. Layton, "File-private POSIX locks", LWN, 19 February 2014. https://lwn.net/Articles/586904/

[lockfileex] Microsoft, `LockFileEx` function. https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-lockfileex

[rust-file] The Rust standard library, `std::fs::File::lock`, `try_lock` and `unlock` (stable since 1.89). https://doc.rust-lang.org/std/fs/struct.File.html#method.lock

[sqlite] SQLite, "How To Corrupt An SQLite Database File", section 2.2, and the comment "Posix Advisory Locking" in `src/os_unix.c`. https://www.sqlite.org/howtocorrupt.html

[git-lockfile] Git, `lockfile.h`. https://github.com/git/git/blob/master/lockfile.h

[cargo-flock] Cargo, `src/util/flock.rs`. https://github.com/rust-lang/cargo/blob/master/src/util/flock.rs

[go-lockedfile] Go, `cmd/go/internal/lockedfile`. https://github.com/golang/go/tree/master/src/cmd/go/internal/lockedfile

[go22315] Go issue 22315, "os: StartProcess ETXTBSY race on Unix systems". https://github.com/golang/go/issues/22315

[go-forklock] Go, `syscall.ForkLock`. https://pkg.go.dev/syscall#ForkLock

[lamport77] Leslie Lamport, “Proving the Correctness of Multiprocess Programs”, IEEE Transactions on Software Engineering SE-3(2):125-143, 1977. doi:10.1109/TSE.1977.229904

[owicki-gries] Susan Owicki and David Gries, “An axiomatic proof technique for parallel programs I”, Acta Informatica 6(4):319-340, 1976. doi:10.1007/BF00268134

[herlihy-wing] Maurice P. Herlihy and Jeannette M. Wing, “Linearizability: a correctness condition for concurrent objects”, ACM Transactions on Programming Languages and Systems 12(3):463-492, 1990. doi:10.1145/78969.78972

[herlihy-waitfree] Maurice Herlihy, “Wait-free synchronization”, ACM Transactions on Programming Languages and Systems 13(1):124-149, 1991. doi:10.1145/114005.102808

[kung-robinson] H. T. Kung and John T. Robinson, “On optimistic methods for concurrency control”, ACM Transactions on Database Systems 6(2):213-226, 1981. doi:10.1145/319566.319567

GDS 0007, discussion catalogs: the journal that makes an update of several files recoverable. GDS 0008, GitHub Pages publishing, and GDS 0009, the native plugin API and registry: the first users of the lock.

Lifecycle event

```
2026-11-23: prediscussion -> discussion. Assigned a permanent number and
opened for discussion.
```

Lifecycle event

```
2026-11-23: discussion -> accepted. The maintainer asked for this record with
the fix of the file lock race, and for its release in Guidedog 1.0.0-beta.15;
the design is implemented.
```

Lifecycle event

```
2026-11-23: accepted -> committed. Ship the implemented file lock in Guidedog
1.0.0-beta.15, as the maintainer asked. gd_plugins::files::lock, used by the
registry, gd-pages, gd-discussion, gd-ssg and the example publisher; its
exhaustive model test and descriptor-copy test; the registry stress test in
crates/gd-plugins/tests/publishers.rs; 13 theorems in docs/gds/proofs checked
by cargo xtask proofs with Lean 4.19; the test binary run 1000 times without
a failure (section Motivation).
```