



GIOLIT | FORMAL VERIFICATION OF C PROGRAMS

Certificate of Formal Verification

Document GIO-6EDFE7EB05CF | issued 2026-10-03 01:24:41 IST (UTC+0530)

PROFESSIONAL LICENSE No. GIOLIT-PRO-2026-000011 licensed to **Acme Ltd.****06_threads_mutex.c is free of the errors listed below.**

For every possible input, no execution of 06_threads_mutex.c (starting at main()) can:

- ✓ fail an assertion (1 place checked)
- ✓ overflow a signed integer (1 place checked)

This is a **mathematical proof** covering all executions, not a test of some of them. The proof itself is part of this certificate (page 3).

Program file	06_threads_mutex.c
Location	/tmp/gv-acme/06_threads_mutex.c
SHA-256	c102c67898a4c1eedc733094777ca67301652b2be0617328c89505b782684c6f
Size	25 lines, entry function main
Issued to	Giolit Labs
Machine ID	527d921416734c6985698c31556b62fc
Machine	Giolit Labs Verification — Ubuntu 24.04.5 LTS (kernel 7.0.14-orbstack-00380-ga7e0a2dc9535); VirtualApple @ 2.50GHz
Verified by	verifier@Giolit Labs Verification
Engine	Giolit Program Verifier v7 7.0.0, build e68a236-modified
Date and time	2026-10-03 01:24:41 IST (UTC+0530) (analysis 0.5 s)
Technique	Inductive invariants (abstract interpretation)
License	Professional license No. GIOLIT-PRO-2026-000011 , licensed to Acme Ltd.

How your client can check this certificate

- 1. Same file.** sha256sum 06_threads_mutex.c must print c102c67898a4c1eedc733094777ca67301652b2be0617328c89505b782684c6f.
- 2. Licensed.** Issued under Giolit professional license No. GIOLIT-PRO-2026-000011, licensed to **Acme Ltd.**
- 3. Same result.** reproduce.sh re-runs the verification.
- 4. Ask us.** Write to business.giolitlabs@gmail.com quoting document GIO-6EDFE7EB05CF.

**Giolit — Formal Verification Services**

Mathematical proof that your software is free of run-time errors. We offer formal verification as a service: proofs of absence of run-time errors (overflow, division by zero, out-of-bounds access, invalid pointers, memory leaks) and of functional properties for C code, certificates for safety standards (DO-178C, ISO 26262, IEC 61508), and review of counterexamples with your engineers.

www.giolit.com business.giolitlabs@gmail.com



www.giolit.com

The program

06_threads_mutex.c (SHA-256 c102c67898a4c1ee...):

```

1  /* Example 6 -- two threads increment a shared counter under a mutex
2  * (Owicki-Gries). Expected: SAFE: after both joins, x == 2. */
3  #include <pthread.h>
4  extern void __assert_fail(const char *, const char *, unsigned int, const char *);
5  void reach_error(void) { __assert_fail("0", "06_threads_mutex.c", 5, "reach_error"); }
6
7  int x = 0;
8  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
9
10 void *worker(void *arg) {
11     pthread_mutex_lock(&m);
12     x = x + 1;
13     pthread_mutex_unlock(&m);
14     return 0;
15 }
16
17 int main(void) {
18     pthread_t a, b;
19     pthread_create(&a, 0, worker, 0);
20     pthread_create(&b, 0, worker, 0);
21     pthread_join(a, 0);
22     pthread_join(b, 0);
23     if (x != 2) reach_error();
24     return 0;
25 }
```

Properties and results

#	Line	Property	CWE	Result	Method
1	12	Signed integer overflow: int32 overflow in +	CWE-190	✓ proven	Inductive invariants (abstract interpretation)
2	23	Assertion: assertion failure (reach_error reached)	CWE-617	✓ proven	Inductive invariants (abstract interpretation)

Verification technique

The program was compiled with the Clang C front end to LLVM IR and translated into a verification model: a control-flow graph whose transitions are exact linear relations over the program's variables, with machine integer ranges, wrap-around, pointers (object, offset), and memory allocation state. Every operation that can fail at run time becomes a *check*: an edge to an error location. A check is **proven** when no execution of the model reaches its error edge; it is **violated** when a concrete execution does. The model over-approximates the program (every real execution is an execution of the model), so a proof on the model is a proof for the program.

Techniques that decided this program:

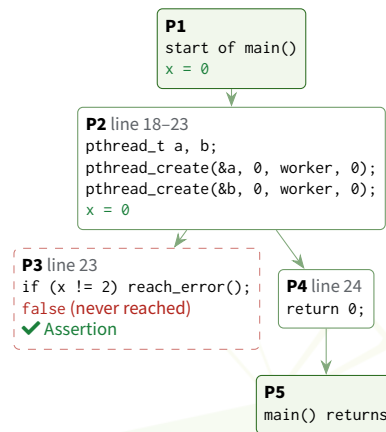
- **Inductive invariants (abstract interpretation).** Inductive invariants were computed for every program point by abstract interpretation over template polyhedra (intervals, octagons and relational directions discovered from the program), with widening and narrowing. The invariant at the failing operation excludes the failure.

Why the annotations are a proof. Each annotation below is an *inductive invariant*: it holds when the program starts, and every transition from an annotated point leads to a point whose annotation holds again. So it holds in every reachable state. At each point where a check could fail, the annotation contradicts the failure condition (points marked **false** are never reached), so no check can fail.

Proof: annotated control-flow graph

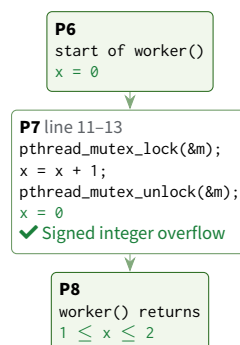
Each box is a program point P_n with its source line and code. The green annotation below the code holds *every time* execution reaches that point, for every input. Arrows are the possible control transfers, labelled with their branch conditions. Checks at a point are listed with ✓ (proven). Dashed boxes are failure branches: their annotation is *false*, so they are never reached. Annotations are shown over the source variables, simplified; the complete machine-checkable annotations are in `proof.json`.

Function `main()`



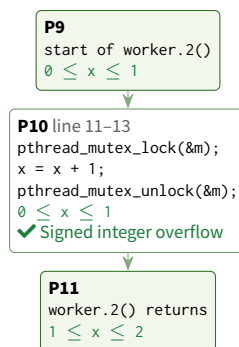
Point	Line	Source	Annotation (holds whenever execution reaches the point)
P1		start of <code>main()</code>	$x = 0$
P2	18-23	<pre>pthread_t a, b; pthread_create(&a, 0, worker, 0); pthread_create(&b, 0, worker, 0); ...</pre>	$x = 0$
P3	23	<code>if (x != 2) reach_error();</code>	<i>false</i> — this point is never reached
P4	24	<code>return 0;</code>	not needed (no check follows this point)
P5		<code>main()</code> returns	not needed (no check follows this point)

Function `worker()`



Point	Line	Source	Annotation (holds whenever execution reaches the point)
P6		start of <code>worker()</code>	$x = 0$
P7	11-13	<pre>pthread_mutex_lock(&m); x = x + 1; pthread_mutex_unlock(&m);</pre>	$x = 0$
P8		<code>worker()</code> returns	$1 \leq x \leq 2$

Function worker.2()



Point	Line	Source	Annotation (holds whenever execution reaches the point)
P9		start of worker.2()	$0 \leq x \leq 1$
P10	11-13	pthread_mutex_lock(&m); x = x + 1; pthread_mutex_unlock(&m);	$0 \leq x \leq 1$
P11		worker.2() returns	$1 \leq x \leq 2$

Assumptions and scope

- **Inputs.** Values returned by functions without a body (for example `__VERIFIER_nondet_int`), the entry function's parameters and external data are arbitrary values of their C types: the result covers every input.
- **Machine model.** The host compiler's data model (x86-64, LP64: int 32 bits, long and pointers 64 bits), two's complement. Signed overflow is reported as an error; unsigned arithmetic wraps as C defines.
- **Entry point.** Executions start at `main()`. Only the source files listed on the first page are analysed.
- **Checks.** all kinds: assertions, signed overflow, division by zero, shifts, array bounds, pointer dereference, free, memory leaks.

Reproducing this result

The bundle delivered with this document contains:

- `06_threads_mutex.c` — the verified source, unchanged
- `certificate.json` — the statement of this result
- `results.json` — per-check results (machine-readable)
- `results.sarif` — SARIF 2.1.0 for code-scanning dashboards and CI
- `proof.json` — the verification model, the complete proof annotations and counterexamples
- `verifier-output.txt` — the verifier's console report
- `reproduce.sh` — the command that produced this result
- `certificate-password.txt` — the password that opens `certificate.pdf`

Command (run in the bundle folder):

```
v7 06_threads_mutex.c
```

About Giolit

We offer formal verification as a service: proofs of absence of run-time errors (overflow, division by zero, out-of-bounds access, invalid pointers, memory leaks) and of functional properties for C code, certificates for safety standards (DO-178C,

ISO 26262, IEC 61508), and review of counterexamples with your engineers. Visit www.giolit.com or write to business.giolitlabs@gmail.com.



Giolit — Formal Verification Services

Mathematical proof that your software is free of run-time errors.

We offer formal verification as a service: proofs of absence of run-time errors (overflow, division by zero, out-of-bounds access, invalid pointers, memory leaks) and of functional properties for C code, certificates for safety standards (DO-178C, ISO 26262, IEC 61508), and review of counterexamples with your engineers.

 www.giolit.com

 business.giolitlabs@gmail.com



www.giolit.com