CON38-C. Preserve thread safety and liveness when using condition variables

Both thread safety and liveness are concerns when using condition variables. The *thread-safety* property requires that all objects maintain consistent states in a multithreaded environment [Lea 2000]. The *liveness* property requires that every operation or function invocation execute to completion without interruption; for example, there is no deadlock.

Condition variables must be used inside a `while` loop. (See CON36-C. Wrap functions that can spuriously wake up in a loop for more information.) To guarantee liveness, programs must test the `while` loop condition before invoking the `cnd_wait()` function. This early test checks whether another thread has already satisfied the condition predicate and has sent a notification. Invoking the `cnd_wait()` function after the notification has been sent results in indefinite blocking.

To guarantee thread safety, programs must test the `while` loop condition after returning from the `cnd_wait()` function. When a given thread invokes the `cnd_wait()` function, it will attempt to block until its condition variable is signaled by a call to `cnd_broadcast()` or to `cnd_signal()`.

The `cnd_signal()` function unblocks one of the threads that are blocked on the specified condition variable at the time of the call. If multiple threads are waiting on the same condition variable, the scheduler can select any of those threads to be awakened (assuming that all threads have the same priority level). The `cnd_broadcast()` function unblocks all of the threads that are blocked on the specified condition variable at the time of the call. The order in which threads execute following a call to `cnd_broadcast()` is unspecified. Consequently, an unrelated thread could start executing, discover that its condition predicate is satisfied, and resume execution even though it was supposed to remain dormant. For these reasons, threads must check the condition predicate after the `cnd_wait()` function returns. A `while` loop is the best choice for checking the condition predicate both before and after invoking `cnd_wait()`.

The use of `cnd_signal()` is safe if each thread uses a unique condition variable. If multiple threads share a condition variable, the use of `cnd_signal()` is safe only if the following conditions are met:

- All threads must perform the same set of operations after waking up, which means that any thread can be selected to wake up and resume for a single invocation of `cnd_signal()`.
- Only one thread is required to wake upon receiving the signal.

The `cnd_broadcast()` function can be used to unblock all of the threads that are blocked on the specified condition variable if the use of `cnd_signal()` is unsafe.

Noncompliant Code Example (`cnd_signal()`)

This noncompliant code example uses five threads that are intended to execute sequentially according to the step level assigned to each thread when it is created (serialized processing). The `current_step` variable holds the current step level and is incremented when the respective thread completes. Finally, another thread is signaled so that the next step can be executed. Each thread waits until its step level is ready, and the `cnd_wait()` function call is wrapped inside a `while` loop, in compliance with CON36-C. Wrap functions that can spuriously wake up in a loop.

```
#include <stdio.h>
#include <threads.h>

enum { NTHREADS = 5 };

mtx_t mutex;
cnd_t cond;

int run_step(void *t) {
  static size_t current_step = 0;
  size_t my_step = *(size_t *)t;

  if (thrd_success != mtx_lock(&mutex)) {
    /* Handle error */
  }

  printf("Thread %zu has the lock\n", my_step);
  while (current_step != my_step) {
    printf("Thread %zu is sleeping...\n", my_step);

    if (thrd_success != cnd_wait(&cond, &mutex)) {
      /* Handle error */
    }

    printf("Thread %zu woke up\n", my_step);
  }
  /* Do processing ... */
  printf("Thread %zu is processing...\n", my_step);
  current_step++;

  /* Signal awaiting task */
  if (thrd_success != cnd_signal(&cond)) {
    /* Handle error */
  }

  printf("Thread %zu is exiting...\n", my_step);

  if (thrd_success != mtx_unlock(&mutex)) {
    /* Handle error */
  }
  return 0;
}
int main(void) {
  thrd_t threads[NTHREADS];
  size_t step[NTHREADS];

  if (thrd_success != mtx_init(&mutex, mtx_plain)) {
    /* Handle error */
  }

  if (thrd_success != cnd_init(&cond)) {
    /* Handle error */
  }

  /* Create threads */
  for (size_t i = 0; i < NTHREADS; ++i) {
    step[i] = i;

    if (thrd_success != thrd_create(&threads[i], run_step,
                                    &step[i])) {
      /* Handle error */
    }
  }

  /* Wait for all threads to complete */
  for (size_t i = NTHREADS; i != 0; --i) {
    if (thrd_success != thrd_join(threads[i-1], NULL)) {
      /* Handle error */
    }
  }

  mtx_destroy(&mutex);
  cnd_destroy(&cond);
  return 0;
}
```

In this example, all threads share a condition variable. Each thread has its own distinct condition predicate because each thread requires `current_step` to have a different value before proceeding. When the condition variable is signaled, any of the waiting threads can wake up.

The following table illustrates a possible scenario in which the liveness property is violated. If, by chance, the notified thread is not the thread with the next step value, that thread will wait again. No additional notifications can occur, and eventually the pool of available threads will be exhausted.

Deadlock: Out-of-Sequence Step Value

Time	Thread # (`my_step`)	`current_step`	Action
0	3	0	Thread 3 executes first time: predicate is `FALSE -> wait()`
1	2	0	Thread 2 executes first time: predicate is `FALSE -> wait()`
2	4	0	Thread 4 executes first time: predicate is `FALSE -> wait()`
3	0	0	Thread 0 executes first time: predicate is `TRUE -> current_step++; cnd_signal()`
4	1	1	Thread 1 executes first time: predicate is `TRUE -> current_step++; cnd_signal()`
5	3	2	Thread 3 wakes up (scheduler choice): predicate is `FALSE -> wait()`
6	—	—	**Thread exhaustion!** No more threads to run, and a conditional variable signal is needed to wake up the others

This noncompliant code example violates the liveness property.

Compliant Solution (`cnd_broadcast()`)

This compliant solution uses the `cnd_broadcast()` function to signal all waiting threads instead of a single random thread. Only the `run_step()` thread code from the noncompliant code example is modified, as follows:

```
#include <stdio.h>
#include <threads.h>

mtx_t mutex;
cnd_t cond;
int run_step(void *t) {
  static size_t current_step = 0;
  size_t my_step = *(size_t *)t;

  if (thrd_success != mtx_lock(&mutex)) {
    /* Handle error */
  }

  printf("Thread %zu has the lock\n", my_step);

  while (current_step != my_step) {
    printf("Thread %zu is sleeping...\n", my_step);

    if (thrd_success != cnd_wait(&cond, &mutex)) {
      /* Handle error */
    }

  printf("Thread %zu woke up\n", my_step);
  }

  /* Do processing ... */
  printf("Thread %zu is processing...\n", my_step);

  current_step++;

  /* Signal ALL waiting tasks */
  if (thrd_success != cnd_broadcast(&cond)) {
    /* Handle error */
  }

  printf("Thread %zu is exiting...\n", my_step);

  if (thrd_success != mtx_unlock(&mutex)) {
    /* Handle error */
  }
  return 0;
}
```

Awakening all threads guarantees the liveness property because each thread will execute its condition predicate test, and exactly one will succeed and continue execution.

Compliant Solution (Using `cnd_signal()` with a Unique Condition Variable per Thread)

Another compliant solution is to use a unique condition variable for each thread (all associated with the same mutex). In this case, `cnd_signal()` wakes up only the thread that is waiting on it. This solution is more efficient than using `cnd_broadcast()` because only the desired thread is awakened.

The condition predicate of the signaled thread must be true; otherwise, a deadlock will occur.

```
#include <stdio.h>
#include <threads.h>

enum { NTHREADS = 5 };

mtx_t mutex;
cnd_t cond[NTHREADS];
```

```
int run_step(void *t) {
  static size_t current_step = 0;
  size_t my_step = *(size_t *)t;

  if (thrd_success != mtx_lock(&mutex)) {
    /* Handle error */
  }

  printf("Thread %zu has the lock\n", my_step);

  while (current_step != my_step) {
    printf("Thread %zu is sleeping...\n", my_step);

    if (thrd_success != cnd_wait(&cond[my_step], &mutex)) {
      /* Handle error */
    }

    printf("Thread %zu woke up\n", my_step);
  }

  /* Do processing ... */
  printf("Thread %zu is processing...\n", my_step);

  current_step++;

  /* Signal next step thread */
  if ((my_step + 1) < NTHREADS) {
    if (thrd_success != cnd_signal(&cond[my_step + 1])) {
      /* Handle error */
    }
  }

  printf("Thread %zu is exiting...\n", my_step);

  if (thrd_success != mtx_unlock(&mutex)) {
    /* Handle error */
  }
  return 0;
}

int main(void) {
  thrd_t threads[NTHREADS];
  size_t step[NTHREADS];

  if (thrd_success != mtx_init(&mutex, mtx_plain)) {
    /* Handle error */
  }

  for (size_t i = 0; i< NTHREADS; ++i) {
    if (thrd_success != cnd_init(&cond[i])) {
      /* Handle error */
    }
  }

  /* Create threads */
  for (size_t i = 0; i < NTHREADS; ++i) {
    step[i] = i;
    if (thrd_success != thrd_create(&threads[i], run_step,
                                    &step[i])) {
      /* Handle error */
    }
  }

  /* Wait for all threads to complete */
  for (size_t i = NTHREADS; i != 0; --i) {
    if (thrd_success != thrd_join(threads[i-1], NULL)) {
      /* Handle error */
    }
  }

  mtx_destroy(&mutex);
  for (size_t i = 0; i < NTHREADS; ++i) {
    cnd_destroy(&cond[i]);
  }
  return 0;
}
```

Compliant Solution (Windows, Condition Variables)

This compliant solution uses a `CONDITION_VARIABLE` object, available on Microsoft Windows (Vista and later):

```
#include <Windows.h>
#include <stdio.h>

CRITICAL_SECTION lock;
CONDITION_VARIABLE cond;

DWORD WINAPI run_step(LPVOID t) {
  static size_t current_step = 0;
  size_t my_step = (size_t)t;

  EnterCriticalSection(&lock);
  printf("Thread %zu has the lock\n", my_step);

  while (current_step != my_step) {
    printf("Thread %zu is sleeping...\n", my_step);

    if (!SleepConditionVariableCS(&cond, &lock, INFINITE)) {
      /* Handle error */
    }

    printf("Thread %zu woke up\n", my_step);
  }

  /* Do processing ... */
  printf("Thread %zu is processing...\n", my_step);

  current_step++;

  LeaveCriticalSection(&lock);

  /* Signal ALL waiting tasks */
  WakeAllConditionVariable(&cond);

  printf("Thread %zu is exiting...\n", my_step);
  return 0;
}

enum { NTHREADS = 5 };

int main(void) {
  HANDLE threads[NTHREADS];

  InitializeCriticalSection(&lock);
  InitializeConditionVariable(&cond);

  /* Create threads */
  for (size_t i = 0; i < NTHREADS; ++i) {
    threads[i] = CreateThread(NULL, 0, run_step, (LPVOID)i, 0, NULL);
  }

  /* Wait for all threads to complete */
  WaitForMultipleObjects(NTHREADS, threads, TRUE, INFINITE);

  DeleteCriticalSection(&lock);

  return 0;
}
```

Risk Assessment

Failing to preserve the thread safety and liveness of a program when using condition variables can lead to indefinite blocking and denial of service (DoS).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CON38-C	Low	Unlikely	Medium	**P2**	**L3**

Automated Detection

Tool	Version	Checker	Description
CodeSonar	7.1p0	**CONCURRENCY.BADFUNC.CNDSIGNAL**	Use of Condition Variable Signal
Helix QAC	2022.2	**C1778, C1779**	
Klocwork	2022.2	**CERT.CONC.UNSAFE_COND_VAR_C**	
Parasoft C/C++test	2022.1	**CERT_C-CON38-a**	Use the 'cnd_signal()' function with a unique condition variable
Polyspace Bug Finder	R2022b	CERT C: Rule CON38-C	Checks for multiple threads waiting on same condition variable (rule fully covered)

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Related Guidelines

Key here (explains table format and definitions)

Taxonomy	Taxonomy item	Relationship
CERT Oracle Secure Coding Standard for Java	THI02-J. Notify all waiting threads rather than a single thread	Prior to 2018-01-12: CERT: Unspecified Relationship

Bibliography

[IEEE Std 1003.1:2013]	XSH, System Interfaces, `pthread_cond_broadcast` XSH, System Interfaces, `pthread_cond_signal`
[Lea 2000]	