CON55-CPP. 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 CON54-CPP. 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 condition_variable::wait() member function. This early test checks whether another thread has already satisfied the condition predicate and has sent a notification. Invoking wait() after the notification has been sent results in indefinite blocking.

To guarantee thread safety, programs must test the while loop condition after returning from wait(). When a given thread invokes wait(), it will attempt to block until its condition variable is signaled by a call to condition_variable::notify_all() or to condition_variable::

notify_one().

The notify_one() member 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 notify_all() member 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 notify_all() 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 wait() function returns. A while loop is the best choice for checking the condition predicate both before and after invoking wait().

The use of notify_one() is safe if each thread uses a unique condition variable. If multiple threads share a condition variable, the use of notify_one() 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 notify_one().
- Only one thread is required to wake upon receiving the signal.

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

Noncompliant Code Example (notify_one())

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 currentStep 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 wait() call is wrapped inside a while loop, in compliance with CON54-CPP. Wrap functions that can spuriously wake up in a loop.
```cpp
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <thread>

std::mutex mutex;
std::condition_variable cond;

void run_step(size_t myStep) {
    static size_t currentStep = 0;
    std::unique_lock<std::mutex> lk(mutex);
    std::cout << "Thread " << myStep << " has the lock" << std::endl;

    while (currentStep != myStep) {
        std::cout << "Thread " << myStep << " is sleeping..." << std::endl;
        cond.wait(lk);
        std::cout << "Thread " << myStep << " woke up" << std::endl;
    }

    // Do processing...
    std::cout << "Thread " << myStep << " is processing..." << std::endl;
    currentStep++;

    // Signal awaiting task.
    cond.notify_one();
    std::cout << "Thread " << myStep << " is exiting..." << std::endl;
}

int main() {
    constexpr size_t numThreads = 5;
    std::thread threads[numThreads];

    // Create threads.
    for (size_t i = 0; i < numThreads; ++i) {
        threads[i] = std::thread(run_step, i);
    }

    // Wait for all threads to complete.
    for (size_t i = numThreads; i != 0; --i) {
        threads[i - 1].join();
    }
}
```

In this example, all threads share a single condition variable. Each thread has its own distinct condition predicate because each thread requires `currentStep` 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**

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread # (my_step)</th>
<th>current_step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>0</td>
<td>Thread 3 executes the first time: the predicate is false -&gt; wait()</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>Thread 2 executes the first time: the predicate is false -&gt; wait()</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>Thread 4 executes the first time: the predicate is false -&gt; wait()</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>Thread 0 executes the first time: the predicate is true -&gt; currentStep++; notify_one()</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>Thread 1 executes the first time: the predicate is true -&gt; currentStep++; notify_one()</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
<td>Thread 3 wakes up (scheduler choice): the predicate is false -&gt; wait()</td>
</tr>
<tr>
<td>6</td>
<td>—</td>
<td>—</td>
<td>Thread exhaustion! There are no more threads to run, and a conditional variable signal is needed to wake up the others.</td>
</tr>
</tbody>
</table>

This noncompliant code example violates the liveness property.
Compliant Solution (\texttt{notify\_all()})

This compliant solution uses \texttt{notify\_all()} to signal all waiting threads instead of a single random thread. Only the \texttt{run\_step()} thread code from the noncompliant code example is modified.

```cpp
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <thread>

std::mutex mutex;
std::condition_variable cond;

void run_step(size_t myStep) {
    static size_t currentStep = 0;
    std::unique_lock<std::mutex> lk(mutex);

    std::cout << "Thread " << myStep << " has the lock" << std::endl;

    while (currentStep != myStep) {
        std::cout << "Thread " << myStep << " is sleeping..." << std::endl;
        cond.wait(lk);
        std::cout << "Thread " << myStep << " woke up" << std::endl;
    }

    // Do processing ...
    std::cout << "Thread " << myStep << " is processing..." << std::endl;
    currentStep++;

    // Signal ALL waiting tasks.
    cond.notify_all();

    std::cout << "Thread " << myStep << " is exiting..." << std::endl;
}

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 \texttt{notify\_one()} 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, \texttt{notify\_one()} wakes up only the thread that is waiting on it. This solution is more efficient than using \texttt{notify\_all()} because only the desired thread is awakened.

The condition predicate of the signaled thread must be true; otherwise, a deadlock will occur.
```cpp
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <thread>

constexpr size_t numThreads = 5;

std::mutex mutex;
std::condition_variable cond[numThreads];

void run_step(size_t myStep) {
  static size_t currentStep = 0;
  std::unique_lock<std::mutex> lk(mutex);

  std::cout << "Thread " << myStep << " has the lock" << std::endl;

  while (currentStep != myStep) {
    std::cout << "Thread " << myStep << " is sleeping..." << std::endl;
    cond[myStep].wait(lk);
    std::cout << "Thread " << myStep << " woke up" << std::endl;
  }

  // Do processing ...
  std::cout << "Thread " << myStep << " is processing..." << std::endl;
  currentStep++;

  // Signal next step thread.
  if ((myStep + 1) < numThreads) {
    cond[myStep + 1].notify_one();
  }

  std::cout << "Thread " << myStep << " is exiting..." << std::endl;
}

// ... main() unchanged ...
```

**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).

<table>
<thead>
<tr>
<th>Rule</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>CON55-CPP</td>
<td>Low</td>
<td>Unlikely</td>
<td>Medium</td>
<td>P2</td>
<td>L3</td>
</tr>
</tbody>
</table>

**Automated Detection**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_CPP-CON55-a</td>
<td>Do not use the 'notify_one()' function when multiple threads are waiting on the same condition variable</td>
</tr>
<tr>
<td>PRQA QA-C++</td>
<td>4.4</td>
<td>5020</td>
<td></td>
</tr>
</tbody>
</table>

**Related Vulnerabilities**

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

**Related Guidelines**

<table>
<thead>
<tr>
<th>Guideline</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CERT Oracle Secure Coding Standard for Java</td>
<td>THI02-J. Notify all waiting threads rather than a single thread</td>
</tr>
<tr>
<td>SEI CERT C Coding Standard</td>
<td>CON38-C. Preserve thread safety and liveness when using condition variables</td>
</tr>
<tr>
<td>SEI CERT C++ Coding Standard</td>
<td>CON54-CPP. Wrap functions that can spuriously wake up in a loop</td>
</tr>
<tr>
<td>Reference</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>IEEE Std 1003.1:2013</td>
<td>XSH, System Interfaces, <code>pthread_cond_broadcast</code></td>
</tr>
<tr>
<td></td>
<td>XSH, System Interfaces, <code>pthread_cond_signal</code></td>
</tr>
<tr>
<td>[Lea 2000]</td>
<td></td>
</tr>
</tbody>
</table>