POS53-C. Do not use more than one mutex for concurrent waiting operations on a condition variable

`pthread_cond_wait()` and `pthread_cond_timedwait()` take a condition variable and locked mutex as arguments. These functions unlock the mutex until the condition variable is signaled and then relock the mutex before returning. While a thread is waiting on a particular condition variable and mutex, other threads may only wait on the same condition variable if they also pass the same mutex as an argument. This requirement is noted in the *Open Group Base Specifications, Issue 6*:

> *As long as at least one thread is blocked on the condition variable. During this time, the effect of an attempt by any thread to wait on that condition variable using a different mutex is undefined.*

It also specifies that `pthread_cond_wait()` *may* fail if:

> *[EINVAL]*
> *The value specified by cond or mutex is invalid.*
> *[EPERM]*
> *The mutex was not owned by the current thread at the time of the call.*

Noncompliant Code Example

In this noncompliant code example, `mutex1` protects `count1` and `mutex2` protects `count2`. A race condition exists between the `waiter1` and `waiter2` threads because they use the same condition variable with different mutexes. If both threads attempt to call `pthread_cond_wait()` at the same time, one thread will succeed and the other thread will invoke undefined behavior.

```
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <assert.h>
#include <unistd.h>
#include <errno.h>

pthread_mutex_t mutex1;
pthread_mutex_t mutex2;
pthread_mutexattr_t attr;
pthread_cond_t cv;

void *waiter1();
void *waiter2();
void *signaler();

int count1 = 0, count2 = 0;
#define COUNT_LIMIT 5

int main() {
  int ret;
  pthread_t thread1, thread2, thread3;

  if ((ret = pthread_mutexattr_init( &attr)) != 0) {
    /* Handle error */
  }

  if ((ret = pthread_mutexattr_settype( &attr, PTHREAD_MUTEX_ERRORCHECK)) != 0) {
    /* Handle error */
  }

  if ((ret = pthread_mutex_init( &mutex1, &attr)) != 0) {
    /* Handle error */
  }

  if ((ret = pthread_mutex_init( &mutex2, &attr)) != 0) {
    /* Handle error */
  }

  if ((ret = pthread_cond_init( &cv, NULL)) != 0) {
    /* handle error */
  }
```

```
  if ((ret = pthread_create( &thread1, NULL, &waiter1, NULL))) {
    /* Handle error */
  }

  if ((ret = pthread_create( &thread2, NULL, &waiter2, NULL))) {
    /* handle error */
  }

  if ((ret = pthread_create( &thread3, NULL, &signaler, NULL))) {
    /* Handle error */
  }

  if ((ret = pthread_join( thread1, NULL)) != 0) {
    /* Handle error */
  }

  if ((ret = pthread_join( thread2, NULL)) != 0) {
    /* Handle error */
  }

  if ((ret = pthread_join( thread3, NULL)) != 0) {
    /* Handle error */
  }

  return 0;
}


void *waiter1() {
  int ret;
  while (count1 < COUNT_LIMIT) {
    if ((ret = pthread_mutex_lock(&mutex1)) != 0) {
      /* Handle error */
    }

    if ((ret = pthread_cond_wait(&cv, &mutex1)) != 0) {
      /* Handle error */
    }

    printf("count1 = %d\n", ++count1);

    if ((ret = pthread_mutex_unlock(&mutex1)) != 0) {
      /* Handle error */
    }
  }

  return NULL;
}

void *waiter2() {
  int ret;
  while (count2 < COUNT_LIMIT) {
    if ((ret = pthread_mutex_lock(&mutex2)) != 0) {
      /* Handle error */
    }

    if ((ret = pthread_cond_wait(&cv, &mutex2)) != 0) {
      /* Handle error */
    }

    printf("count2 = %d\n", ++count2);

    if ((ret = pthread_mutex_unlock(&mutex2)) != 0) {
      /* Handle error */
    }
  }

  return NULL;
}
```

```
void *signaler() {
  int ret;
  while ((count1 < COUNT_LIMIT) || (count2 < COUNT_LIMIT)) {
    sleep(1);
    printf("signaling\n");
    if ((ret = pthread_cond_signal(&cv)) != 0) {
      /* Handle error */
    }
  }

  return NULL;
}
```

Implementation Details: Linux

When the system is built on the following platform,

> *Red Hat Enterprise Linux Client release 5.5 (Tikanga)*
> *kernel 2.6.18*
> *gcc 4.3.5 with the --D_GNU_SOURCE flag*

the preceding code works as expected. `waiter1` and `waiter2` increment the variable once they are signaled, and the correct mutex is acquired after `pthread_cond_wait` returns in each thread.

The man page for `pthread_cond_wait` on this configuration says that it *may* fail with a return value of `EINVAL` if "different mutexes were supplied for concurrent `pthread_cond_timedwait()` or `pthread_cond_wait()` operations on the same condition variable." However, this does not happen.

Implementation Details: OS X

When the system is built on the following platform,

> *OS X 10.6.4 (Snow Leopard)*
> *gcc 4.2.1*

`pthread_cond_wait()` returns `EINVAL` if it is called when another thread is waiting on the condition variable with a different mutex. This approach is arguably better because it forces the coder to fix the problem instead of allowing reliance on undefined behavior.

The man page for `pthread_cond_wait()}] simply says that {{EINVAL` will be returned if "the value specified by cond or the value specified by mutex is invalid," but it doesn't say what *invalid* means.

Compliant Solution

This problem can be solved either by always using the same mutex whenever a particular condition variable is used or by using separate condition variables, depending on how the code is expected to work. This compliant solution uses the same-mutex solution:

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

pthread_mutex_t mutex1; /* Initialized as PTHREAD_MUTEX_ERRORCHECK */
pthread_cond_t cv;
int count1 = 0, count2 = 0;
#define COUNT_LIMIT 5

void *waiter1() {
  int ret;
  while (count1 < COUNT_LIMIT) {
    if ((ret = pthread_mutex_lock(&mutex1)) != 0) {
      /* Handle error */
    }

    if ((ret = pthread_cond_wait(&cv, &mutex1)) != 0) {
      /* Handle error */
    }

    printf("count1 = %d\n", ++count1);

    if ((ret = pthread_mutex_unlock(&mutex1)) != 0) {
      /* Handle error */
    }
  }

  return NULL;
}

void *waiter2() {
  int ret;
  while (count2 < COUNT_LIMIT) {
    if ((ret = pthread_mutex_lock(&mutex1)) != 0) {
      /* Handle error */
    }

    if ((ret = pthread_cond_wait(&cv, &mutex1)) != 0) {
      /* Handle error */
    }

    printf("count2 = %d\n", ++count2);

    if ((ret = pthread_mutex_unlock(&mutex1)) != 0) {
      /* Handle error */
    }
  }

  return NULL;
}
```

Risk Assessment

Waiting on the same condition variable with two different mutexes could cause a thread to be signaled and resume execution with the wrong mutex locked. It could lead to unexpected program behavior if the same shared data were simultaneously accessed by two threads.

The severity is medium because improperly accessing shared data could lead to data integrity violation. Likelihood is probable because in such an implementation, an error code would not be returned, and remediation cost is high because detection and correction of this problem are both manual.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
POS53-C	Medium	Probable	High	**P4**	**L3**

Automated Detection

Tool	Version	Checker	Description
Parasoft C/C++test	10.4.2	**CERT_C-POS53-a**	Do not use more than one mutex for concurrent waiting operations on a condition variable

Bibliography

[Open Group 2004]	`pthread_cond_timedwait()/pthread_cond_wait()`