CON34-C. Declare objects shared between threads with appropriate storage durations

Accessing the automatic or thread-local variables of one thread from another thread is implementation-defined behavior and can cause invalid memory accesses because the execution of threads can be interwoven within the constraints of the synchronization model. As a result, the referenced stack frame or thread-local variable may no longer be valid when another thread tries to access it. Shared static variables can be protected by thread synchronization mechanisms.

However, automatic (local) variables cannot be shared in the same manner because the referenced stack frame's thread would need to stop executing, or some other mechanism must be employed to ensure that the referenced stack frame is still valid. Do not access automatic or thread-local objects from a thread other than the one with which the object is associated. See DCL30-C. Declare objects with appropriate storage durations for information on how to declare objects with appropriate storage durations when data is not being shared between threads.

Noncompliant Code Example (Automatic Storage Duration)

This noncompliant code example passes the address of a variable to a child thread, which prints it out. The variable has automatic storage duration. Depending on the execution order, the child thread might reference the variable after the variable's lifetime in the parent thread. This would cause the child thread to access an invalid memory location.

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

int child_thread(void *val) {
    int *res = (int*)val;
    printf("Result: %d\n", *res);
    return 0;
}

void create_thread(thrd_t *tid) {
    int val = 1;
    if (thrd_success != thrd_create(tid, child_thread, &val)) {
        /* Handle error */
    }
}

int main(void) {
    thrd_t tid;
    create_thread(&tid);
    if (thrd_success != thrd_join(tid, NULL)) { /* Handle error */
        return 0;
    }
}
```

Noncompliant Code Example (Automatic Storage Duration)

One practice is to ensure that all objects with automatic storage duration shared between threads are declared such that their lifetime extends past the lifetime of the threads. This can be accomplished using a thread synchronization mechanism, such as `thrd_join()`. In this code example, `val` is declared in `main()`, where `thrd_join()` is called. Because the parent thread waits until the child thread completes before continuing its execution, the shared objects have a lifetime at least as great as the thread.
```c
#include <threads.h>
#include <stdio.h>

int child_thread(void *val) {
    int *result = (int *)val;
    printf("Result: %d\n", *result); /* Correctly prints 1 */
    return 0;
}

void create_thread(thrd_t *tid, int *val) {
    if (thrd_success != thrd_create(tid, child_thread, val)) {
        /* Handle error */
    }
}

int main(void) {
    int val = 1;
    thrd_t tid;
    create_thread(&tid, &val);
    if (thrd_success != thrd_join(tid, NULL)) {
        /* Handle error */
    }
    return 0;
}

However, the C Standard, 6.2.4 paragraphs 4 and 5 [ISO/IEC 9899:2011], states:

- The result of attempting to indirectly access an object with thread storage duration from a thread other than the one with which the object is associated is implementation-defined. . .
- The result of attempting to indirectly access an object with automatic storage duration from a thread other than the one with which the object is associated is implementation-defined.

Therefore this example relies on implementation-defined behavior and is nonportable.

Compliant Solution (Static Storage Duration)

This compliant solution stores the value in an object having static storage duration. The lifetime of this object is the entire execution of the program; consequently, it can be safely accessed by any thread.
```c
#include <threads.h>
#include <stdio.h>

int child_thread(void *v) {
    int *result = (int *)v;
    printf("Result: %d\n", *result); /* Correctly prints 1 */
    return 0;
}

void create_thread(thrd_t *tid) {
    static int val = 1;
    if (thrd_success != thrd_create(tid, child_thread, &val)) {
        /* Handle error */
    }
}

int main(void) {
    thrd_t tid;
    create_thread(&tid);
    if (thrd_success != thrd_join(tid, NULL)) {
        /* Handle error */
    }
    return 0;
}
```

**Compliant Solution (Allocated Storage Duration)**

This compliant solution stores the value passed to the child thread in a dynamically allocated object. Because this object will persist until explicitly freed, the child thread can safely access its value.

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

int child_thread(void *val) {
    int *result = (int *)val;
    printf("Result: %d\n", *result); /* Correctly prints 1 */
    return 0;
}

void create_thread(thrd_t *tid, int *value) {
    *value = 1;
    if (thrd_success != thrd_create(tid, child_thread, value)) {
        /* Handle error */
    }
}

int main(void) {
    thrd_t tid;
    int *value = (int *)malloc(sizeof(int));
    if (!value) {
        /* Handle error */
    }
    create_thread(&tid, value);
    if (thrd_success != thrd_join(tid, NULL)) {
        /* Handle error */
    }
    free(value);
    return 0;
}
```

**Noncompliant Code Example (Thread-Specific Storage)**

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

int child_thread(void *v) {
    int *result = (int *)v;
    printf("Result: %d\n", *result); /* Correctly prints 1 */
    return 0;
}

void create_thread(thrd_t *tid) {
    static int val = 1;
    if (thrd_success != thrd_create(tid, child_thread, &val)) {
        /* Handle error */
    }
}

int main(void) {
    thrd_t tid;
    create_thread(&tid);
    if (thrd_success != thrd_join(tid, NULL)) {
        /* Handle error */
    }
    return 0;
}
```
In this noncompliant code example, the value is stored in thread-specific storage of the parent thread. However, because thread-specific data is available only to the thread that stores it, the `child_thread()` function will set `result` to a null value.

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

static tss_t key;

int child_thread(void *v) {
    void *result = tss_get(*(tss_t *)v);
    printf("Result: %d\n", *(int *)result);
    return 0;
}

int create_thread(void *thrd) {
    int *val = (int *)malloc(sizeof(int));
    if (val == NULL) {
        /* Handle error */
    }
    *val = 1;
    if (thrd_success != tss_set(key, val)) {
        /* Handle error */
    }
    if (thrd_success != thrd_create((thrd_t *)thrd,
            child_thread, &key)) {
        /* Handle error */
    }
    return 0;
}

int main(void) {
    thrd_t parent_tid, child_tid;

    if (thrd_success != tss_create(&key, free)) {
        /* Handle error */
    }
    if (thrd_success != thrd_create(&parent_tid, create_thread,
            &child_tid)) {
        /* Handle error */
    }
    if (thrd_success != thrd_join(parent_tid, NULL)) {
        /* Handle error */
    }
    if (thrd_success != thrd_join(child_tid, NULL)) {
        /* Handle error */
    }
    tss_delete(key);
    return 0;
}
```

Compliant Solution (Thread-Specific Storage)

This compliant solution illustrates how thread-specific storage can be combined with a call to a thread synchronization mechanism, such as `thrd_join()`. Because the parent thread waits until the child thread completes before continuing its execution, the child thread is guaranteed to access a valid live object.
```c
#include <threads.h>
#include <stdio.h>
#include <stdlib.h>

static tss_t key;

int child_thread(void *v) {
    int *result = v;
    printf("Result: %d\n", *result); /* Correctly prints 1 */
    return 0;
}

int create_thread(void *thrd) {
    int *val = (int *)malloc(sizeof(int));
    if (val == NULL) {
        /* Handle error */
    }
    *val = 1;
    if (thrd_success != tss_set(key, val)) {
        /* Handle error */
    }
    /* ... */
    void *v = tss_get(key);
    if (thrd_success != thrd_create((thrd_t *)thrd,
         child_thread, v)) {
        /* Handle error */
    }
    return 0;
}

int main(void) {
    thrd_t parent_tid, child_tid;

    if (thrd_success != tss_create(&key, free)) {
        /* Handle error */
    }
    if (thrd_success != thrd_create(&parent_tid, create_thread,
         &child_tid)) {
        /* Handle error */
    }
    if (thrd_success != thrd_join(parent_tid, NULL)) {
        /* Handle error */
    }
    if (thrd_success != thrd_join(child_tid, NULL)) {
        /* Handle error */
    }
    tss_delete(key);
    return 0;
}
```

This compliant solution uses pointer-to-integer and integer-to-pointer conversions, which have implementation-defined behavior. (See INT36-C. Converting a pointer to integer or integer to pointer.)

Compliant Solution (Thread-Local Storage, Windows, Visual Studio)

Similar to the preceding compliant solution, this compliant solution uses thread-local storage combined with thread synchronization to ensure the child thread is accessing a valid live object. It uses the Visual Studio–specific __declspec(thread) language extension to provide the thread-local storage and the WaitForSingleObject() API to provide the synchronization.
```
#include <Windows.h>
#include <stdio.h>

DWORD WINAPI child_thread(LPVOID v) {
    int *result = (int *)v;
    printf("Result: %d\n", *result);  /* Correctly prints 1 */
    return NULL;
}

int create_thread(HANDLE *tid) {
    /* Declare val as a thread-local value */
    __declspec(thread) int val = 1;
    *tid = create_thread(NULL, 0, child_thread, &val, 0, NULL);
    return *tid == NULL;
}

int main(void) {
    HANDLE tid;

    if (create_thread(&tid)) {
        /* Handle error */
    }

    if (WAIT_OBJECT_0 != WaitForSingleObject(tid, INFINITE)) {
        /* Handle error */
    }
    CloseHandle(tid);
    return 0;
}

Noncompliant Code Example (OpenMP, parallel)

It is important to note that local data can be used securely with threads when using other thread interfaces, so the programmer need not always copy data into nonlocal memory when sharing data with threads. For example, the `shared` keyword in The OpenMP® API Specification for Parallel Programming[OpenMP] can be used in combination with OpenMP's threading interface to share local memory without having to worry about whether local automatic variables remain valid.

In this noncompliant code example, a variable `j` is declared outside a `parallel` #pragma and not listed as a private variable. In OpenMP, variables outside a `parallel` #pragma are shared unless designated as private.

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

int main(void) {
    int j = 0;
    #pragma omp parallel
    {
        int t = omp_get_thread_num();
        printf("Running thread - %d\n", t);
        for (int i = 0; i < 5050; i++) {
            j++;  /* j not private; could be a race condition */
        }  
        printf("Just ran thread - %d\n", t);
        printf("loop count %d\n", j);
    }
    return 0;
}

Compliant Solution (OpenMP, parallel, private)

In this compliant solution, the variable `j` is declared outside of the `parallel` #pragma but is explicitly labeled as `private`:

```
Risk Assessment

Threads that reference the stack of other threads can potentially overwrite important information on the stack, such as function pointers and return addresses. The compiler may not generate warnings if the programmer allows one thread to access another thread's local variables, so a programmer may not catch a potential error at compile time. The remediation cost for this error is high because analysis tools have difficulty diagnosing problems with concurrency and race conditions.

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>CON34-C</td>
<td>Medium</td>
<td>Probable</td>
<td>High</td>
<td>P4</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_C-CON34-a</td>
<td>Declare objects shared between POSIX threads with appropriate storage durations</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>4926, 4927, 4928</td>
<td>Enforced by QAC</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>Taxonomy</th>
<th>Taxonomy item</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>CERT C Secure Coding Standard</td>
<td>DCL30-C. Declare objects with appropriate storage durations</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
</tbody>
</table>

Bibliography


[OpenMP] The OpenMP® API Specification for Parallel Programming