Get proactive with limitations in COBOL multithreading

Judith M. Myerson (jmyerson@verizon.net)
Systems engineer and architect

Often businesses and government agencies need to run COBOL programs faster by running them as multithreaded applications. In reality, COBOL multithreading comes with some limitations. In this article, the author explains the pitfalls of limitations and how to handle them.

Overview

Multithreading allows each task of a well-designed program to operate as independent of others—nearly at the same time. Performance improves when multiple threads run on multiple CPUs (dual or quad-core). A multithreaded application works just as well on a single-CPU system but without the added speed.

Over the years, I have seen how COBOL programs evolved from the time they ran in a single thread—slowly by today's standards—on a bygone IBM mainframe to the present day's time of running in multiple threads on much faster, smaller machines. The late Navy Rear Admiral Grace Hopper, the mother of COBOL, who was also the author's thesis advisor, would have loved to hear about IBM COBOL for AIX.

The nuts and bolts of running COBOL programs in multiple threads are that you must compile all COBOL programs with the THREAD compiler option. A same program can have separate threads—for example, one thread for a task of the program, a second thread for the second task of the same program, and so on.

One thing you need to watch out for is that some COBOL applications depend on subsystems or other applications and may not work with some language elements. These dependencies and language element restrictions impose limitations on COBOL programs. By getting around these limitations, you can take proactive steps to get the programs to reap the benefits of running in multithreading.

In this article, I give you the steps you should take to avoid becoming reactive to handling the limitations with COBOL multithreading:

1. Watch out for unwanted language elements
2. Work with the scopes of language elements
3. Get ready for compilation
4. Compile, link and run programs
First step: Watch out for unwanted language elements

When you write COBOL programs for a multithreaded environment, choose appropriate linkage statements and language elements including statements, special registers, and clauses that make up the logic of a COBOL program. Watch out for the language elements that will not work with multithreading. If you compile the program containing unwanted elements with the THREAD compiler option, they are flagged as errors.

Table 1 below shows what language elements to avoid that are not supported by the THREAD compiler option. These language elements are categorized as obsolete in Standard COBOL 85 and are not part of Standard COBOL 2002.

**Table 1. Language elements the THREAD compiler option doesn't support**

<table>
<thead>
<tr>
<th>Language element</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER</td>
<td>Statement</td>
</tr>
<tr>
<td>DEBUG-ITEM</td>
<td>Special register</td>
</tr>
<tr>
<td>GO TO without procedure name</td>
<td>Statement</td>
</tr>
<tr>
<td>INITIAL phrase in PROGRAM-ID</td>
<td>Clause</td>
</tr>
<tr>
<td>RERUN</td>
<td>Clause</td>
</tr>
<tr>
<td>Segmentation module</td>
<td>Module</td>
</tr>
<tr>
<td>STOP literal</td>
<td>Statement</td>
</tr>
<tr>
<td>STOP RUN</td>
<td>Statement</td>
</tr>
<tr>
<td>USE FOR DEBUGGING</td>
<td>Declarative</td>
</tr>
</tbody>
</table>

When you find those unwanted language elements, you may need to replace them with elements with similar functions that the THREAD compiler option supports. For some replacements, you may need to change the program logic or linkage statements.

Here are some examples of how you can handle unsupported language elements:

Replace the ALTER statement with the EVALUATE statement. The obsolete ALTER statement provides the same function as EVALUATE. However, EVALUATE provides more, better options that encourage the use of structured programming. The ALTER statement changes the transfer point specified in the GO TO statement. The EVALUATE statement does more than that, it can be used instead of a series of IF statements to several conditions and specify a different action for each. You can use EVALUATE to build a case structure.

If you find the GO TO statement is missing a reference to a procedure, the program does not compile correctly. Instead, use the PERFORM procedure-name-1 THRU procedure-name-2. If you are tempted to use the GO TO statement with a procedure name, consider the EVALUATE statement as an alternative option.

The STOP RUN statement closes all files defined in any of the programs in the run unit. There is no COBOL function that effectively does a STOP RUN in a threaded environment. If you
need this behavior, consider calling the C exit function from your COBOL program and using _twzCOBOLTerm after the runtime termination exit.

Consider GOBACK or EXIT PROGRAM as another way of terminating the program. The GOBACK statement specifies the logical end of a called program or invoked method. This returns you to the caller of the program. When you use GOBACK from the first program (not always the main program) in a thread, the program is terminated. Use EXIT PROGRAM as you would GOBACK, except from a main program where it has no effect.

When you do not preinitialize a COBOL environment to perform a storage cleanup, and the COBOL run-time determines there are no active COBOL programs in the runt unit, the COBOL process for terminating a run unit (including closing all open COBOL files) is performed on the GOBACK from the first program of the thread. This determination can be made if all COBOL programs within the run unit have returned to their callers through GOBACK or EXIT PROGRAM.

Do not use a CANCEL statement to replace the EXIT PROGRAM statement. If the CANCEL statement names a program that is active to any thread, the execution fails. The named program must be inactive.

In place of the USE FOR DEBUGGING, use the -g compiler option to prompt the compiler to generate debug information for the source code.

**Second step: Work with language elements in two scopes**

Because your COBOL programs can run as separate threads within a process, a language element can be interpreted in two different scopes: run-unit scope or program invocation instance scope. These two types of scope are important in determining where an item can be referenced and how long the item persists in storage.

In a multithreaded environment, a COBOL run unit is the portion of the process that includes threads that have been actively executing COBOL programs. While the COBOL run unit runs, the language element persists and is available to other programs within this thread. It continues until no COBOL program is active in the execution stack for any of the threads. For example, a called COBOL program contains a GOBACK statement (a language element) and returns control to a C program. Within the run unit, COBOL programs can call non-COBOL programs and vice versa.

Within a thread, control is transferred between separate COBOL and non-COBOL programs. For example, a COBOL program can call another COBOL program or a C program. Each separately called program is a program invocation instance. Those instances of a particular program can exist in multiple threads within a given process. The language element persists only within a particular program invocation.

Table 2 summarizes the scope of various COBOL language elements that can be used with multithreading. They include special registers, statements, and DATA DIVISION sections.
Table 2. Scope of COBOL language elements

<table>
<thead>
<tr>
<th>Language element</th>
<th>Referenced from</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDRESS-OF special register</td>
<td>Same as associated record</td>
<td>Program invocation instance</td>
</tr>
<tr>
<td>LENGTH of special register</td>
<td>Same as associated identifier</td>
<td>Same as associated identifier</td>
</tr>
<tr>
<td>LINAGE-COUNTER special register</td>
<td>Run unit</td>
<td>Run unit</td>
</tr>
<tr>
<td>LINKAGE-SECTION data</td>
<td>Run unit</td>
<td>Based on scope of underlying data</td>
</tr>
<tr>
<td>LOCAL-STORAGE data</td>
<td>Within the thread</td>
<td>Program invocation instance</td>
</tr>
<tr>
<td>RETURN-CODE</td>
<td>Run unit</td>
<td>Program invocation instance</td>
</tr>
<tr>
<td>SORT-CONTROL, SORT-CORE-SIZE, SORT-RETURN, TALLY special registers</td>
<td>Run unit</td>
<td>Program invocation instance</td>
</tr>
<tr>
<td>WHEN-COMPILED special register</td>
<td>Run unit</td>
<td>Run unit</td>
</tr>
<tr>
<td>WORKING-STORAGE data</td>
<td>Run unit</td>
<td>Run unit</td>
</tr>
</tbody>
</table>

Special registers

Special registers are reserved words that name storage areas generated by the compiler. Their primary use is to store information produced through specific COBOL features. Each such storage area has a fixed name and must not be defined within the program.

Make sure there is enough storage area for special registers. Always clean up the storage area to be allocated to multiple threads when you run the program.

For programs compiled with the THREAD options (as well as programs with the RECURSIVE attribute in the PROGRAM-ID paragraph), storage for the following special registers is allocated on a per-invocation basis:

- ADDRESS-OF
- RETURN-CODE
- SORT-CONTROL
- SORT-CORE-SIZE
- SORT-FILE-SIZE
- SORT-MESSAGE
- SORT-MODE-SIZE
- SORT-RETURN
- TALLY

Except for the ADDRESS-OF and LINAGE-COUNTER special registers, these special registers can be referenced from run unit. They can also span for the lifetime of a program as run units and are reset to their initial value on each program or method entry. For the first call to a program after a cancel of that program, or for a method invocation, the compiler initializes the fields of these special registers to their initial values. Values set in ADDRESS-OF special registers persist only for the span of the particular program or method invocation.
A separate LINAGE-COUNTER special register is generated for each FD entry that contains a LINAGE clause. When more than one special register is generated, you must qualify each reference to a LINAGE-COUNTER with its related file name.

The value in LINAGE-COUNTER at any given time is the line number at which the device is positioned within the current page. LINAGE-COUNTER can be referred to in procedure division statements; it must not be modified by them.

**Storage data**

In the DATA DIVISION, all files in the FILE SECTION have storage statically allocated and remained for the life of the run unit. How data items are allocated and initialized varies depending on whether the items are in the WORKING-STORAGE SECTION or LOCAL-STORAGE SECTION.

When a program is invoked, the WORKING-STORAGE associated with the program is allocated. Any data items that have VALUE clauses are initialized to the appropriate value at that time. For the duration of the run unit, WORKING-STORAGE items persist in their last-used state.

All data defined in the LOCAL-STORAGE SECTION have storage allocated each time a program is called and de-allocated when the program ends. If you specify a VALUE clause for a LOCAL-STORAGE item, the item is initialized to that value on each call or invocation. If a VALUE clause is not specified, the initial value of the item is undefined. The LINKAGE SECTION describes data from another program.

**Third step: Get ready for compilation**

To run COBOL programs in more than one thread, you must compile all of the COBOL programs in the application with the THREAD compiler option. You cannot mix programs compiled using THREAD compiler option and programs compiled using the default NOTHREAD compiler option in the same application.

To prepare for compilation of COBOL programs with multiple threads, you need to:

- Clean up storage
- Avoid CICS environment
- Work with recursive calls
- Make linkage data accessible

**Clean up storage**

If your program initiates multiple COBOL threads, do not assume that COBOL programs (for example, your C program calls COBOL programs to carry out input and output of data) clean up their environment, such as releasing storage no longer needed.

For your application to control the COBOL cleanup, preinitialize the COBOL environment. You can do this by using the COBOL preinitialization interface of your program. When the files are closed, you make room for storage to be allocated for multiple threads.
Avoid CICS environment

Make sure the environment you compile the program in is not CICS. You cannot run multithreaded applications in the CICS environment. In the CICS environment, you can run a COBOL program in a single thread that has been compiled with the THREAD option, and that is part of an application that has no multiple threads or PL/I tasks.

Work with recursive calls

To make a recursive call, you must either code the RECURSIVE clause (or attribute) in the PROGRAM-ID paragraph of the recursively called program or specify the THREAD compiler option. Because you must code the programs in a multithreaded application as recursive, you cannot code nested programs. You must specify the RECURSIVE clause only on the outermost program of a compilation unit.

Make linkage data accessible

If your program has the RECURSIVE attribute or is compiled with the THREAD compiler option, data that is defined in the LINKAGE SECTION is not accessible on subsequent invocations of the program. To address a record in the LINKAGE SECTION, pass an argument to the program and specify the record in an appropriate position in the USING phrase in the program.

Fourth step: Compile, link, and run programs

You can combine your multithreaded COBOL programs with C programs and Language Environment-enabled assembler programs in the same run unit when those programs are also appropriately coded for multithreaded execution.

A multithreading example is one of the best ways to show you how to use commands to compile, link, and execute a program. Table 3 below lists three program examples: a C main program and two COBOL programs.

Table 3. Programs to compile and run

<table>
<thead>
<tr>
<th>Program name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thrcob.c</td>
<td>A C program that creates two COBOL threads, waits for them and then exits</td>
</tr>
<tr>
<td>subd.cbl</td>
<td>A COBOL program that is run by the thread created by thrcob.c</td>
</tr>
<tr>
<td>subd.cbl</td>
<td>A second COBOL program that is run by the thread created by thrcob.c</td>
</tr>
</tbody>
</table>

Make sure the C program is properly designed, otherwise you might create excessive threads (more than two threads) that would take up much space needed for other tasks. Make sure a COBOL program does not have nested program.

At the AIX shell prompt, it takes three commands to compile the programs, one command to generate an executable main program and run it.
When you compile a COBOL program, use the cob2_r command for non-CICS applications for which you need the THREAD compiler. You can compile object-oriented (OO) clients and classes by using the cob2_j command. This command causes the THREAD option to be included as a default invocation option. Do not use the cob2 command. Always check for compilation errors before you generate and run an executable program.

Create compiled programs

Listing 1 below shows the three lines of commands you should put in an AIX shell prompt to compile a C main program and the two COBOL programs, assuming the source codes are error-free.

Listing 1. Sample compile commands

```
xlc_r -c thecob.c
cob2_r -c -g -qTHREAD subd.cbl
cob2_r -c -g -qTHREAD sube.cbl
```

Obviously the -c option stands for compiling programs but do not link them. The -q option in the command for the COBOL programs passes the THREAD compiler option to compiler. The -g option prompts the compiler to generate debug information for the source code.

Run executable module

After compilation of the programs come out clean, listing 2 below shows two lines of commands. The first command generates an executable module by linking the programs. The second command runs the module.

Listing 2. Sample compile commands

```
cob2_r -o thrcob thecob.o subd.o sube.o
thecob
```

The -o option applies to linking programs. It names thrcob as the executable module and generates it by linking thrcob.o, subd.o and sube.o. If the -o option is not used, the name of the executable module defaults to thrcob.out.

Conclusion

Handling limitations with COBOL multithreading requires planning ahead of time to resolve the issues of removing unwanted language elements, and changing the program logic to include language elements that the THREAD compiler option supports. Developers should communicate with programming teams on working with language elements, preparing for compilation, and compiling, linking and executing programs.
Resources

- Evaluate COBOL for AIX
- Read more on multithreading in IBM COBOL for AIX Programming Guide Version 4.1.
- Get more details on language syntax in IBM COBOL for AIX Language Reference Guide
- Want to try out COBOL for AIX version 4.1.0? Download the 60-day trail.
- Evaluate other AIX and UNIX downloads and products that suit you best.
- Try out IBM software for free. Download a trial version, log into an online trial, work with a product in a sandbox environment, or access it through the cloud. Choose from over 100 IBM product trials.
About the author

Judith M. Myerson

Judith M. Myerson is a systems architect and engineer. Her areas of interest include enterprise-wide systems, middleware technologies, database technologies, cloud computing, threshold policies, service level agreements, industries, network management, security, RFID technologies, presentation management, and project management. She was one of the first COBOL programmers to write and run a resource security program.

© Copyright IBM Corporation 2011
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)