IBM

System i

Programming
Multithreaded applications

Version 6 Release 1

IBM

System i

Programming
Multithreaded applications

Version 6 Release 1

> **Note**
>
> Before using this information and the product it supports, read the information in "Notices," on page 73.

Contents

Multithreaded applications

A *thread* is the path taken, the steps performed, and the order in which the steps are performed by a running program.

The term thread is short for *thread of control*. A thread runs code from its starting location in an ordered, predefined sequence for a given set of inputs. You can use multiple threads to improve application performance by running different application tasks simultaneously.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

What's new for V6R1

Read about new or significantly changed information for the Multithreaded applications topic collection.

New method to create data that is private to a thread

You can now use thread local storage (TLS), a new method to create data that is private to a thread. Thread local storage is similar to thread-specific storage, except that thread local storage is implemented by the operating system, whereas thread-specific storage is implemented with APIs. As a result, thread local storage is faster and easier to use than thread-specific storage. ILE C, ILE C++, and ILE RPG compilers support thread local storage.

For more information about thread local storage, see the following topics:
- "Thread-private data, thread-specific data, and thread local storage" on page 4
- "Storage usage and threaded applications" on page 28
- "Data that is private to a thread" on page 49

How to see what's new or changed

To help you see where technical changes have been made, the information center uses:
- The ≫ image to mark where new or changed information begins.
- The ≪ image to mark where new or changed information ends.

In PDF files, you might see revision bars (|) in the left margin of new and changed information.

To find other information about what's new or changed this release, see the Memo to users.

PDF file for Multithreaded applications

You can view and print a PDF file of this information.

To view or download the PDF version, select Multithreaded applications (about 734 KB).

Saving PDF files

To save a PDF on your workstation for viewing or printing:
1. Right-click the PDF link in your browser.
2. Click the option that saves the PDF locally.
3. Navigate to the directory in which you want to save the PDF.

4. Click **Save**.

Downloading Adobe Reader

You need Adobe® Reader installed on your system to view or print these PDFs. You can download a free

copy from the Adobe Web site (www.adobe.com/products/acrobat/readstep.html) .

Related reference

"Related information for Multithreaded applications" on page 71
Product manuals, Web sites, and other information center topic collections contain information that
relates to the Multithreaded applications topic collection. You can view or print any of the PDF files.

Threads on i5/OS

All programs have at least one thread, referred to as the *initial thread*. In a program with multiple threads,
each thread runs its code independently of the other threads in the program.

A *process* is the container for the memory and resources of the program. On IBM® System i® platforms, a
job represents a process. Each process has at least one thread (task) in which the program runs its code.
The first thread in a process is referred to as the initial thread. Some processes are capable of supporting
additional threads, which are called *secondary threads*.

The concepts that are described here pertain to all programming languages. For details about how each
language enables a concept, refer to the programmer's guide for that specific language.

Thread models

There are different models of thread usage: user threads and kernel threads.

In the user thread model, all program threads share the same process thread. In the user thread model,
the thread application programming interfaces (APIs) enforce the scheduling policy and decide when a
new thread runs. The scheduling policy allows only one thread to be actively running in the process at a
time. The operating system kernel is only aware of a single task in the process.

In the kernel thread model, kernel threads are separate tasks that are associated with a process. The
kernel thread model uses a preemptive scheduling policy in which the operating system decides which
thread is eligible to share the processor. In a kernel thread model, there is a one-to-one mapping between
program threads and process threads. The i5/OS® operating system supports a kernel thread model.

Some platforms support a combination of these two thread models, commonly referred to as an MxN
thread model. Each process has M user threads that share N kernel threads. The user threads are
scheduled on top of the kernel threads. The system allocates resources only to the more "expensive"
kernel threads.

Thread program models

There are different models for creating multithread programs.

One of the key differences between the i5/OS operating system and other operating systems is the ability
of i5/OS to support a call-return program model. On other platforms, if one program wants to call
another program, it must start a second process to run the second program, or must replace the first
program with the second. Starting another process to call a program is expensive in terms of startup time
and system resources. To avoid this expense, programmers group commonly used functions into
dynamically linked libraries (DLLs). Whenever a program needs a service that a DLL provides, the
program loads the DLL and calls the function that provides the needed service.

Although the i5/OS call-return program model is supported for multithreaded programs, you are strongly encouraged to use service programs or DLLs that run in the caller's activation group. If you perform ports of multithreaded programs from other platforms, you will naturally employ service programs in your ports.

Although it is not a requirement, all programs used in a multithreaded application should be Integrated Language Environment® (ILE) programs. The use of the original program model (OPM) in multithreaded programs requires special considerations. You must evaluate the thread safety of existing ILE or OPM programs if multithreaded programs call them.

> **Related reference**
>
> "Threads considerations for OPM language" on page 61
> Original program model (OPM) programs are not threadsafe. You should migrate OPM programs to Integrated Language Environment (ILE) and make them threadsafe before a multithreaded application calls them.

Jobs and job resources

A *job* is a container for storage and other resources, and it cannot run by itself. *Job resources* are resources that are accessible by all threads within the job.

Every job has two basic kinds of storage associated with it.

Data: Data is where all of the program variables are stored. It is broken down into storage for global and static variables (static), thread local storage (TLS), storage for dynamically allocated storage (heap), and storage for variables that are local to the function (automatic). The storage for program variables is allocated from the activation group into which the program was activated. Static storage is accessible to all threads that are running in the activation group. Thread local storage and automatic storage are local to the thread. Access to heap storage depends on the storage type for the heap-storage pointer.

Stack: The stack contains data about the program or procedure call flow in a thread. The stack, along with thread local storage and automatic storage, is allocated for each thread created. When in use by a thread, the stack, thread local storage, and automatic storage are considered to be thread resources. When the thread ends, these resources return to the process for subsequent use by another thread.

Job resources are as follows:
- Coded character set identifier (CCSID)
- Locales
- Environment variables
- File descriptors
- Files that are opened with a job scope
- Signal action vector
- Signal timers
- The current working directory

All threads running in an activation group can share activation group resources, such as static and heap storage. When a thread changes a job resource, the new value is immediately visible to all threads. For example, changing the signal handling action in one thread effectively changes the signal handling action in all threads.

Thread-private data, thread-specific data, and thread local storage

Threads cannot share certain resources. Data that threads cannot share between themselves are called *thread-private data*. Data that threads cannot share between themselves can be defined using either *thread-specific data* or *thread local storage*.

The i5/OS operating system defines the following resources as thread-private data.

- Thread identifier

  The unique integral number that can be used to identify the thread.

- Priority

  i5/OS supports specification of a thread priority, which determines the relative importance of one thread to other threads in the job. The *thread priority* is defined to be a delta value to the job's priority. i5/OS adds the thread's priority to the job's priority, and the result can never exceed the job's priority. If you adjust the job's priority, the thread's priority is adjusted relative to the new job priority. The default thread priority value is zero. This results in a thread that has the same priority as the job.

- Security information

  Security information, including user and group profiles, is maintained on a per-thread basis. When a thread creates a new thread, the new thread inherits the security information from the thread that created it.

- Library list

  Library list information is maintained on a per-thread basis. When a thread creates a new thread, the new thread inherits the library list information from the thread that created it.

- Signal blocking mask

  The signal blocking mask identifies the set of asynchronous signals to be blocked from delivery to the thread. When a thread creates a new thread, the new thread inherits the signal blocking mask of the thread that created it.

- Call stack

  The call stack contains data about the program flow or procedure call flow in a thread. The stack, along with automatic storage, is allocated for each thread created.

- Automatic storage

  Automatic storage is for variables that are local to the function.

- Errno variable

  This program variable is used to return the result of a C or POSIX system call. Errno is a function call that returns the most recent result for a function call in the thread.

Thread-specific data is different from thread-private data. The threads implementation defines the thread-private data, while the application defines the thread-specific data. Threads do not share thread-specific storage (it is specific to a thread), but all functions within that thread can access it. Typically, a key indexes thread-specific storage. The key is a global value that is visible to all threads. It is used to retrieve the thread-specific value of the storage associated with that key.

Data that is defined using thread local storage is similar to thread-specific data. Threads do not share thread local storage, but all functions within a thread can access the storage for that thread. Thread local storage is different from thread-specific data in that thread local storage is managed by the language environment, whereas thread-specific data is managed by the application.

Multithreaded program startup commands

To call a multithreaded Integrated Language Environment (ILE) program, the job in which the program is called must be capable of supporting multiple threads (multithread-capable). Multithreaded Java™ programs can be called in most jobs (some system jobs restrict the types of programs that can be called).

In the i5/OS kernel threads support, only a subset of the supported job types can create threads. Interactive and communication jobs do no provide multithread-capable support.

The parameter **Allow Multithread (ALWMLTTHD)**, which is located in the Create Job Description (CRTJOBD) and Change Job Description (CHGJOBD) commands, controls whether the job supports multiple threads. i5/OS examines all job types except communications jobs and interactive jobs for the **ALWMLTTHD** parameter. The Add Prestart Job Entry (ADDPJE) and Change Prestart Job Entry (CHGPJE) subsystem description job entry commands use the **ALWMLTTHD** setting to control whether jobs started through the entry are capable of supporting multiple threads.

The spawn() API, Spawn Process, supports a flag field in the inheritance structure that controls whether the child process can support multiple threads. This flag field, SPAWN_SETTHREAD_NP, is a non-standard, i5/OS platform-specific extension to the inheritance structure. The spawn() API is the only programming method that can start a batch immediate or prestart job that is capable of supporting multiple threads. You can write a SPAWN command, similar to the CALL command, to simplify calling multithreaded programs. An example SPAWN command is available for your use and modification as part of i5/OS option 7, the i5/OS Example Tools Library, QUSRTOOLS.

Activation groups and threads

All programs and service programs are activated within a substructure of a job that is called an activation group. This substructure contains the resources necessary to run the programs, including static and heap storage, and temporary data management resources.

In a job capable of multithreaded operation, two or more threads can share an activation group. A thread can run programs that were activated in different activation groups. The system does not keep a list of threads that run in an activation group nor does it keep a list of activation groups in a thread. Ending an activation group with active threads in it can produce unpredictable results including an abnormal end of the process. To avoid these problems, any action that ends an activation group in a secondary thread causes the system to end the job in an orderly fashion. In other words, the system ends all threads in the job, calls exit routines and C++ destructors in the initial thread, and finally closes the files.

When you work with activation groups, be aware of the following considerations:

- Return from activation groups

  A program that is created with **ACTGRP(*NEW)** gets a new activation group every time it is called. When the program returns, the new activation group ends. In a secondary thread, returning from a program that was created with **ACTGRP(*NEW)** will end the job.

  Both default and named activation groups are persistent. The default activation group is created when a job starts and is destroyed only when the job ends. Named activation groups are created when they are first needed in a job. A normal return from the named activation group leaves the activation group in last-used state, but does not delete the activation group. Hence, in a job capable of multithreaded operations, a normal return from the default or named activation group does not cause the job to end.

  Using exit() or abort() in any activation group in a secondary thread always ends the job.

- Unhandled exceptions

  If an exception has not been handled by the time it has percolated to the control boundary and the control boundary is a program entry procedure (PEP or main entry point), the multithread-capable job is ended.

- Reclaim activation group (RCLACTGRP)

  Jobs capable of multithreaded operation allow this command to be called only in the initial thread on the job. If secondary user threads are present in the job, this command sends a CPF180B escape message to the caller of the command.

  Related concepts

"Thread safety" on page 28

A function is threadsafe if you can start it simultaneously in multiple threads within the same process. A function is threadsafe only if all the functions it calls are also threadsafe.

Communications considerations for multithreaded programming

The only threadsafe communications protocol supported on the i5/OS operating system is sockets.

The use of sockets is subject to the following restrictions:

- Socket APIs:

  The majority of the sockets interfaces are threadsafe, but most of the network routines are not due to the use of static storage. These routines have been replaced with threadsafe _r counterparts. For example, you should replace calls to gethostbyaddr() with gethostbyaddr_r(). The _r routines are compatible with the UNIX® definition. All _r functions reside in the existing service program QSOSRV2.

- AnyNet® using sockets:

  AnyNet using sockets is considered threadsafe and is supported in multithread programs. However, this support has not undergone any formal testing.

Database considerations for multithreaded programming

When you use databases in multithreaded programs, be aware of these considerations.

- Data definition language (DDL)

  Many of the database configuration, administration, and setup type interfaces are threadsafe. Database operations that are threadsafe include: Create File, Add Member, Delete File, and Remove Member. Refer to CL Reference or use the Display Command (DSPCMD) command to determine if a command is threadsafe. The online help information for that command lists any necessary conditions for threadsafety that apply to a command.

- Database record I/O

  The database protects the I/O operations for the duration of the operation (read, update, insert, or delete). When you share an open instance of a file among threads, you must serialize access to the I/O feedback areas and I/O buffers to see valid information in these areas. These areas are under application control, and the database cannot protect them after the database operation has been completed.

  An example is a read operation. If thread 1 is in the process of a read operation and thread 2 performs any I/O operation against the same open instance, thread 2 waits until thread 1 has completed the reading. The result of the read operation in thread 1 is placed in the I/O buffer. When control returns to thread 1, thread 2 begins its I/O operation. Without serialization, thread 2 can change the information in the I/O buffer before thread 1 can view the result.

  If threads do not share an open instance of a file, no serialization is required.

- Distributed files

  Access to distributed database files used with the DB2® Multisystem and access to distributed data management (DDM) files of type *SNA are not threadsafe. Multithreaded jobs deny access to database files of these types. These file types cannot be made threadsafe because intersystem communications function (ICF) files and the entire Systems Network Architecture (SNA) layer are not threadsafe. If an attempt is made to open one of these file types, a CPF4380 message (`Open attributes not valid in a multithreaded process`) is sent to the function attempting to open the file.

- Trigger programs

  You can fire trigger programs in a multithreaded job. The same threadsafety restrictions apply to trigger programs as to any other code that runs in a multithreaded job. Parameters on the Add Physical File Trigger (ADDPFTRG) command allow for specification of the trigger's threadsafety status and the action to take if the trigger is fired in a multithreaded job.

- Format selector programs

For logical files with multiple formats, the use of format selector programs is not threadsafe. You should not use format selector programs in a multithreaded job.

- Stored procedures

  DB2 Structured Query Language (SQL) stored-procedure support provides a way for an SQL application to define and call an external program though SQL statements. Stored procedures can be called in a multithreaded job. The same thread safety restrictions apply to stored procedures as to any other code that runs in a multithreaded job. Unlike trigger programs, there is no way of specifying the stored procedure's thread safety status and the action to take if the stored procedure is called in a multithreaded job.

- SQL statements

  The use of DDL SQL statements might not be threadsafe. Data manipulation language (DML) statements are threadsafe.

- Server mode for SQL

  Using the server mode for SQL is the preferred method for accessing databases with multithreaded applications. A job can use the server mode for SQL in order to manage multiple database connections and transactions. When the application is using server mode for SQL, the i5/OS operating system uses the connections in the job as a more encapsulated representation of the current database context than previously permitted in i5/OS. It allows for connections to a database by multiple users, multiple connections to a database by the same or different users, and the existence of multiple, independent transactions by connections to a database.

  Use one of the following mechanisms to activate server mode for SQL before data access occurs in the application:

  - Use the Open Database Connectivity (ODBC) API, SQLSetEnvAttr() and set the SQL_ATTR_SERVER_MODE attribute to SQL_TRUE before doing any data access.
  - Use the Change Job API, QWTCHGJB() and set the 'Server mode for Structured Query Language' key before doing any data access.
  - Use Java Database Connectivity (JDBC) to access the database. JDBC automatically uses server mode to preserve required semantics of JDBC.

  Server mode for SQL behavior

  - For embedded SQL, each thread in a job is a separate transaction that can be committed or rolled back, even if there are multiple connections within that thread.
  - For ODBC, call level interface (CLI), and JDBC, each connection handle represents a stand-alone connection to the database and can be committed and used as a separate entity.

  The CLI for SQL is threadsafe.

- Commitable transactions

  The introduction of threads does not change the scope of committable transactions. You can scope committable units of work to either a job-level commitment definition or an activation group-level commitment definition. A thread commit or rollback operation commits or rolls-back all operations done under the commitment definition. If you want an application where each thread (or group of threads) has a separate committable transaction, you must either use server mode for SQL or manage these transactions with separate activation groups.

  Related concepts

  Control Language

  Threads

  SQL call level interface

  Related reference

  "Database record I/O and thread safety" on page 58
  The I/O feedback area of a database file can be used to communicate results of I/O operations to the record I/O user. A typical application might have problems with thread safety because of the nature of the feedback area.

Using the i5/OS registration facility, you can define exit points for functions in an application and register programs that run at those exit points.

Database transactions in the i5/OS operating system are scoped to the job or the activation group.

Example: Working with local SQL databases in multithreaded Pthread programs

This example demonstrates how Embedded SQL can be used within a threaded ILE C program.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/********************************************************/
/* Testcase: SQLEXAMPLE                                 */
/*                                                      */
/* Function:                                            */
/* Demonstrate how Embedded SQL can be used within a    */
/* threaded ILE C program.  This program creates and    */
/* populates an SQL Table with data.  Then, threads are */
/* created which each open a cursor and read all the    */
/* data from the table.  A semaphore is used to show    */
/* how threads execution can be controlled.             */
/*                                                      */
/* To compile program:                                  */
/* CRTSQLCI OBJ(QGPL/SQLEXAMPLE) SRCFILE(QGPL/QCSRC)    */
/* COMMIT(*NONE) RDB(*NONE) OBJTYPE(*MODULE)            */
/* OUTPUT(*PRINT) DBGVIEW(*SOURCE)                      */
/*                                                      */
/* To bind program:                                     */
/* CRTPGM PGM(QGPL/SQLEXAMPLE)                          */
/* MODULE(QGPL/SQLEXAMPLE) ACTGRP(*CALLER)              */
/*                                                      */
/* To invoke program:                                   */
/* SPAWN QGPL/SQLEXAMPLE                                */
/*                                                      */
/********************************************************/
#define  _MULTI_THREADED
#include <pthread.h>
#include <sys/sem.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

static int semid;
static struct sembuf op_try1[1] = {0,0,0};
#define MAXTHREADS 2

void *threadrtn(void *parm);

int main(int argc, char **argv)

{
  int           rc;
  int           *status;
  pthread_t     thids[MAXTHREADS];
  EXEC SQL BEGIN DECLARE SECTION;
  int           i, j;
  char          insert[200];
  EXEC SQL END   DECLARE SECTION;

  EXEC SQL  INCLUDE SQLCA;
  EXEC SQL  INCLUDE SQLDA;

  /* create a new semaphore */
```

```
    semid=semget(IPC_PRIVATE,1,S_IRUSR|S_IWUSR);

    printf("\nsemaphore created\n");
    rc=semctl(semid, 0, SETVAL, 1);

    printf("semaphore inited\n");
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL CREATE TABLE QGPL/SQLEXMP (COL1 INT,COL2 INT);

    printf("SQL table created\n");
    EXEC SQL WHENEVER SQLERROR GO TO :mainerror;

    for (i=1,j=100;i<10;i++,j++) {
        (void) sprintf(insert, "INSERT INTO QGPL/SQLEXMP \
                       VALUES(%d, %d)", i, j);
        EXEC SQL EXECUTE IMMEDIATE :insert;
    }

    printf("Table primed with data\n");
    for (i=0;i<MAXTHREADS;i++)  {
        pthread_create(&thids[i], NULL, threadrtn, NULL);
    }
    printf("Threads created\n");

    rc=semctl(semid, 0, SETVAL, 0);

    printf("Threads turned loose\n");
    for (i=0;i<MAXTHREADS;i++) {
        pthread_join(thids[i], &status);
    }

    printf("Threads joined\n");
    return;

mainerror:
    printf("ERROR: sqlcode = %d sqlstate = %d\n", SQLCODE, SQLSTATE);
}

/**********************************************************/
/* This thread will do the following:                     */
/*   - Declare a cursor for the example table             */
/*   - Block on a semaphore until initial thread           */
/*     is ready for us to run                             */
/*   - Open the cursor                                    */
/*   - Fetch data one row at a time in a loop until       */
/*     End of File is reached or an error occurs          */
/*   - Close the cursor and return                        */
/**********************************************************/
void *threadrtn(void *parm)
{
 EXEC SQL  INCLUDE SQLCA;
 EXEC SQL  INCLUDE SQLDA;

 EXEC SQL BEGIN DECLARE SECTION;
 long HV1, HV2;
 EXEC SQL END   DECLARE SECTION;

 EXEC SQL WHENEVER SQLERROR GO TO :thderror;
 EXEC SQL WHENEVER NOT FOUND GO TO :thdeof;

 EXEC SQL DECLARE C1 CURSOR FOR SELECT * FROM QGPL/SQLEXMP;

 /* block on semaphore */
 semop(semid,&op_try1[0],1);

 EXEC SQL OPEN C1;
 printf("thid:%.8x %.8x: cursor open\n",pthread_getthreadid_np());
```

```
/* Loop until End of File (EOF) */
 for (;;) {
    EXEC SQL FETCH C1 INTO :HV1, :HV2;
    printf("thid:%.8x %.8x: fetch done... COL1=%d COL2=%d\n",
           pthread_getthreadid_np(), HV1, HV2);
 }

thderror:
 printf("thid:%.8x %.8x: sqlcode = %d sqlstate = %d\n",
        pthread_getthreadid_np(), SQLCODE, SQLSTATE);
 EXEC SQL CLOSE C1;
 return;

thdeof:
 printf("thid:%.8x %.8x: Done!\n",
        pthread_getthreadid_np());
 return;
}



Testcase output:
 semaphore created
 semaphore inited
 SQL table created
 Table primed with data
 Threads created
 Threads turned loose
 thid:00000000 00000022: cursor open
 thid:00000000 00000023: cursor open
 thid:00000000 00000023: fetch done... COL1=1 COL2=100
 thid:00000000 00000022: fetch done... COL1=1 COL2=100
 thid:00000000 00000023: fetch done... COL1=2 COL2=101
 thid:00000000 00000022: fetch done... COL1=2 COL2=101
 thid:00000000 00000023: fetch done... COL1=3 COL2=102
 thid:00000000 00000022: fetch done... COL1=3 COL2=102
 thid:00000000 00000023: fetch done... COL1=4 COL2=103
 thid:00000000 00000022: fetch done... COL1=4 COL2=103
 thid:00000000 00000023: fetch done... COL1=5 COL2=104
 thid:00000000 00000022: fetch done... COL1=5 COL2=104
 thid:00000000 00000023: fetch done... COL1=6 COL2=105
 thid:00000000 00000022: fetch done... COL1=6 COL2=105
 thid:00000000 00000023: fetch done... COL1=7 COL2=106
 thid:00000000 00000022: fetch done... COL1=7 COL2=106
 thid:00000000 00000023: fetch done... COL1=8 COL2=107
 thid:00000000 00000022: fetch done... COL1=8 COL2=107
 thid:00000000 00000023: fetch done... COL1=9 COL2=108
 thid:00000000 00000022: fetch done... COL1=9 COL2=108
 thid:00000000 00000023: Done!
 thid:00000000 00000022: Done!
 Threads joined
```

Data management considerations for multithreaded programming

When you manage data in a multithreaded program, be aware of these considerations.

- File open operations

  Only integrated file system stream files, printer files, distributed data management (DDM) files of type
  *IP, and local database files can be opened in a multithread-capable job. Attempts to open *FILE objects
  other than print files, type *IP DDM files, or local database files cause a CPF4380 escape message to be
  sent to the function that opens the file. The CPF4380 escape message signals that open attributes are
  not valid in a multithreaded process. Local database files include physical and logical files. Loosely
  coupled parallel (LCP) files are not local and send the CPF4380 escape message when a function tries
  to open the file. Local database files also do not include save files (*SAVF), which are device files, or

other communications files. You cannot open a printer file by specifying SPOOL(*NO). If you specify SPOOL(*NO) when trying to open the print file, the CPF4380 escape message is sent to the function that tried to open the print file.

Multithread-capable jobs allow shared opens. However, an open data path (ODP) cannot be shared between threads. If a file is opened SHARE(*YES) and OPNSCOPE(*ACTGRPDFN) is specified, any subsequent shared open within the same thread that is running in the same activation group can share the ODP. If a file is opened SHARE(*YES) with OPNSCOPE(*JOB) specified, any subsequent shared open of the file within the same thread can share the ODP.

- File overrides

  Multithreaded applications can only issue overrides in their initial thread. Any attempt to issue an override in a secondary thread returns the CPF180C escape message. Only overrides with job-level and activation group-level scope affect secondary threads; overrides of call-level scope are not visible. Overrides of all scope levels can affect initial threads. You can use three different types of file override commands: Override with Database File (OVRDBF), Override with Printer File (OVRPRTF), and Override with Message File (OVRMSGF).

  You can delete overrides in the initial thread, but not in secondary threads, by calling the Delete Override (DLTOVR) command. DLTOVR returns the CPF180C escape message if thrown against a secondary thread.

- Reclaim resources commands

  None of the commands that reclaim resources are threadsafe because the i5/OS operating system does not track resources. The operating system cannot identify which resources are in use by which threads. You cannot call the Reclaim Resources (RCLRSC) and Reclaim Activation Group (RCLACTGRP) commands in secondary threads. RCLRSC and RCLACTGRP can be called in the initial thread of a multithread-capable job (ALWMLTTHD(*YES)) if secondary user threads are not present. If secondary user threads are present in the job, these commands send a CPF180B escape message to the caller of the command.

  Related reference

  "Printer file considerations for multithreaded programming" on page 12
  Only printer files opened with SPOOL(*YES) are allowed in a job that is capable of multithread operations.

File system considerations for multithreaded programming

When you work with file systems in multithreaded programs, be aware of these considerations.

- Threadsafe file systems

  The "root" (/), QOpenSys, User-Defined (UDFS), QNTC, QSYS.LIB, QOPT, NFS, and QFileSvr.400 file systems are threadsafe and do not have any restrictions. The QDLS file system is not threadsafe. If you try to use an integrated file system application programming interface (API) or command on a file or file descriptor that represents an object in a file system that is not threadsafe, it fails with ENOTSAFE. This failure occurs only when multiple threads are in a job.

  An attempt to spawn a program from a job that has multiple threads fails with ENOTSAFE if you inherit the current working directory or open file descriptors that represent files in a file system that is not threadsafe.

- Integrated file system APIs

  All of the integrated file system APIs are threadsafe when directed to an object that resides in a threadsafe file system. If you do not know the file system where an object resides, you can query the path to see if the threadsafe integrated file system interfaces can safely access it. You can use the pathconf(), fpathconf(), statvfs(), and fstatvfs() APIs to determine whether a path or file descriptor refers to an object in one of the threadsafe file systems.

  Related reference

  Integrated file system APIs

Printer file considerations for multithreaded programming

Only printer files opened with SPOOL(*YES) are allowed in a job that is capable of multithread operations.

Unspooled opens of printer files fail with a CPF4380 (Open attributes not valid in a multithreaded process) escape message sent to the function that requested the open.

Most SNA character string (SCS), Intelligent Printer Data Stream (IPDS), Advanced Function Printing data stream (AFPDS), LINE, and USERASCII write operations are threadsafe. Write operations that are not threadsafe include those that use SCS graphics, those that the System/36™ Environment uses, those generated by the User Interface Manager (UIM), and those that use System/36 PRPQ subroutines. Page and line counts might be inaccurate because those counters cannot be atomically updated.

> **Related reference**
>
> "Data management considerations for multithreaded programming" on page 10
> When you manage data in a multithreaded program, be aware of these considerations.

i5/OS thread management

The concepts described here pertain to all programming languages.

For details about how each language enables a concept, refer to the programmer's guide for that specific language.

> **Related information**
>
> Pthread APIs

Thread attributes

Thread attributes are thread characteristics that affect the behavior of the thread. Different attributes are available depending on the programming language and the API that you are using.

Methods for using an attribute and its effect on the thread depend on how the programming language and the API externalize the thread attribute to your application. You can set the thread attributes at the time you start a thread or change them after the thread is actively running.

Some common thread attributes and their effects are as follows:

- Priority

  The priority attribute affects the amount of processing time that the system gives the thread before letting another thread or process interrupt it.

- Stack size

  The stack size attribute affects the number of functions that a thread can call before the thread fails due to insufficient stack space.

- Name

  The name attribute affects the ability to debug or track the actions of a thread through your application.

- Thread group

  The thread group attribute affects the ability to easily manage more than one thread at a time.

- Detach state

  The detach state attribute affects how you reclaim or leave active resources associated with a thread when a thread ends.

- Scheduling policy

  The scheduling policy attribute affects how the threads are scheduled within the system or within the application. This relates to thread priority.

- Inherit scheduling

  The inherit scheduling attribute affects how the priority of the thread is determined by the system.

Example: Setting a thread attribute in a Pthread program

This example shows how to set the *detach state* thread attribute in a Pthread program.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
Filename: ATEST10.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST10
 Create a default thread attributes object
 Set the detach state thread attribute
 Create a thread using the new attributes
 Destroy thread attributes object
 Join now fails because the detach state attribute was changed
 Entered the thread
 Main completed
*/
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define checkResults(string, val) {                \
 if (val) {                                        \
   printf("Failed with %d at %s", val, string);  \
   exit(1);                                        \
 }                                                 \
}

void *theThread(void *parm)
{
   printf("Entered the thread\n");
   return NULL;
}

int main(int argc, char **argv)
{
  pthread_attr_t        attr;
  pthread_t             thread;
  int                   rc=0;

  printf("Enter Testcase - %s\n", argv[0]);

  printf("Create a default thread attributes object\n");
  rc = pthread_attr_init(&attr);
  checkResults("pthread_attr_init()\n", rc);

  printf("Set the detach state thread attribute\n");
  rc = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
  checkResults("pthread_attr_setdetachstate()\n", rc);

  printf("Create a thread using the new attributes\n");
  rc = pthread_create(&thread, &attr, theThread, NULL);
  checkResults("pthread_create()\n", rc);

  printf("Destroy thread attributes object\n");
  rc = pthread_attr_destroy(&attr);
  checkResults("pthread_attr_destroy()\n", rc);

  printf("Join now fails because the detach state attribute was changed\n");
  rc = pthread_join(thread, NULL);
```

```
  if (rc==0) {
     printf("Unexpected results from pthread_join()\n");
     exit(1);
  }
  sleep(2);

  printf("Main completed\n");
  return 0;
}
```

Example: Setting a thread attribute in a Java program

This example shows how to set the *name* thread attribute in a Java program.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
FileName: ATEST10.java

The output of this example is as follows:
Entered the testcase
Create a thread
Set some thread attributes
Start the thread
Wait for the thread to complete
Entered the thread: "theThread"
Testcase completed
*/

import java.lang.*;

public class ATEST10 {

   static class theThread extends Thread {
      public void run() {
         System.out.print("Entered the thread: \"" + getName() +
                          "\"\n");
      }
   }

   public static void main(String argv[]) {
      System.out.print("Entered the testcase\n");

      System.out.print("Create a thread\n");
      theThread t = new theThread();

      System.out.print("Set some thread attributes\n");
      t.setName("theThread");

      System.out.print("Start the thread\n");
      t.start();

      System.out.print("Wait for the thread to complete\n");
      try {
         t.join();
      }
      catch (InterruptedException e) {
         System.out.print("Join interrupted\n");
      }

      System.out.print("Testcase completed\nj");
      System.exit(0);
   }

}
```

Starting a thread

When your application creates a thread, the system initializes a thread object, control structures, and runtime support. These allow the new thread to use language constructs and system services safely.

In addition, you might need to initialize the application data and parameters that the new thread uses before starting the new thread.

When you create a thread, the system assigns the thread a unique thread identifier. The thread identifier is an integer value that you can use when you debug, trace, or perform other administrative activities on the thread. You typically do not use this thread identifier to directly manipulate the thread.

Most APIs that create a thread return a thread object or handle that represents the newly created thread. You can use this thread object to manipulate the new thread, or use this thread object as a synchronization object to wait for the thread to finish its processing.

Example: Starting a thread in a Pthread program

This example shows how to start and pass parameters to a thread in a Pthread program.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
Filename: ATEST11.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST11
 Create/start a thread with parameters
 Wait for the thread to complete
 Thread ID 0000000c, Parameters: 42 is the answer to "Life, the Universe and Everything"
 Main completed
*/
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define checkResults(string, val) {              \
 if (val) {                                      \
   printf("Failed with %d at %s", val, string);  \
   exit(1);                                       \
 }                                                \
}
typedef struct {
   int      threadParm1;
   char     threadParm2[124];
} threadParm_t;

void *theThread(void *parm)
{
   pthread_id_np_t      tid;
   threadParm_t *p = (threadParm_t *)parm;
   tid = pthread_getthreadid_np();
   printf("Thread ID %.8x, Parameters: %d is the answer to \"%s\"\n",
          tid.intId.lo, p->threadParm1, p->threadParm2);
   return NULL;
}

int main(int argc, char **argv)
{
  pthread_t             thread;
  int                   rc=0;
  threadParm_t          *threadParm;

  printf("Enter Testcase - %s\n", argv[0]);
```

```
   threadParm = (threadParm_t *)malloc(sizeof(threadParm));
   threadParm->threadParm1 = 42;
   strcpy(threadParm->threadParm2, "Life, the Universe and Everything");

   printf("Create/start a thread with parameters\n");
   rc = pthread_create(&thread, NULL, theThread, threadParm);
   checkResults("pthread_create()\n", rc);

   printf("Wait for the thread to complete\n");
   rc = pthread_join(thread, NULL);
   checkResults("pthread_join()\n", rc);

   printf("Main completed\n");
   return 0;
}
```

Example: Starting a thread using a Java program

This program shows how to start a thread in a Java program.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
FileName: ATEST11.java

The output of this example is as follows:
 Entered the testcase
 Create a thread with parameters
 Start the thread
 Wait for the thread to complete
 Thread Parameters: 42 is the answer to "Life, the Universe and Everything"
 Testcase completed
*/
import java.lang.*;


public class ATEST11 {

   static class theThread extends Thread {
      int             threadParm1;
      String          threadParm2;

      public theThread(int i, String s) {
         threadParm1 = i;
         threadParm2 = s;
      }
      public void run() {
         System.out.print("Thread Parameters: " + String.valueOf(threadParm1) +
                        " is the answer to \"" + threadParm2 + "\"\n");
      }
   }

   public static void main(String argv[]) {
      System.out.print("Entered the testcase\n");

      System.out.print("Create a thread with parameters\n");
      theThread t = new theThread(42, "Life, the Universe and Everything");

      System.out.print("Start the thread\n");
      t.start();

      System.out.print("Wait for the thread to complete\n");
      try {
         t.join();
      }
      catch (InterruptedException e) {
```

```
        System.out.print("Join interrupted\n");
    }

    System.out.print("Testcase completed\nj");
    System.exit(0);
    }

}
```

Ending a thread

When a thread has completed its processing, it takes an action to end itself and release system resources for use by other threads.

Some APIs require the application to explicitly release resources associated with the thread when the application ends. Other threads implementations (like Java) garbage collect and clean up resources when it is appropriate to do so.

A thread can end in several ways. From a performance perspective, the best way to end a thread is to return from the initial routine that was called when the thread was started. APIs that create a thread typically provide mechanisms for ending the thread when returning from the initial routine is not an option.

Some APIs also support exception mechanisms. Exception mechanisms for ending a thread result in the thread ending when it takes an exception that is not handled. An example might be a Java exception that is thrown by a thread.

See your language-specific documentation for details on exceptions or other ways a thread can end itself.

Related reference

"Threads as synchronization primitives" on page 44
Threads themselves can be used as synchronization primitives when one thread specifically waits for another thread to complete.

Example: Ending a thread in a Pthread program

This example shows how to end a thread in a Pthread program.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
Filename: ATEST12.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST12
 Create/start a thread
 Wait for the thread to complete, and release its resources
 Thread: End with success
 Check the thread status
 Main completed
*/
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define checkResults(string, val) {               \
 if (val) {                                        \
   printf("Failed with %d at %s", val, string); \
   exit(1);                                        \
 }                                                 \
}

const int THREADFAIL = 1;
```

```
const int THREADPASS = 0;

void *theThread(void *parm)
{
   printf("Thread: End with success\n");
   pthread_exit(__VOID(THREADPASS));
   printf("Thread: Did not expect to get here!\n");
   return __VOID(THREADFAIL);
}

int main(int argc, char **argv)
{
  pthread_t            thread;
  int                  rc=0;
  void                 *status;

  printf("Enter Testcase - %s\n", argv[0]);

  printf("Create/start a thread\n");
  rc = pthread_create(&thread, NULL, theThread, NULL);
  checkResults("pthread_create()\n", rc);

  printf("Wait for the thread to complete, and release its resources\n");
  rc = pthread_join(thread, &status);
  checkResults("pthread_join()\n", rc);

  printf("Check the thread status\n");
  if (__INT(status) != THREADPASS) {
     printf("The thread failed\n");
  }

  printf("Main completed\n");
  return 0;
}
```

Example: Ending a thread using a Java program

This example shows how to end a thread in a Java program.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
FileName: ATEST12.java

The output of this example is as follows:
 Entered the testcase
 Create a thread
 Start the thread
 Wait for the thread to complete
 Thread: End with success
 Check the thread status
 Testcase completed
*/
import java.lang.*;


public class ATEST12 {

   static class theThread extends Thread {
      public final static int THREADFAIL = 1;
      public final static int THREADPASS = 0;
      int        _status;

      public int status() {
         return _status;
      }
      public theThread() {
```

```
      _status = THREADFAIL;
   }
   public void run() {
      System.out.print("Thread: End with success\n");
      _status = THREADPASS;
      /* End the thread without returning    */
      /* from its initial routine            */
      stop();
      System.out.print("Thread: Didn't expect to get here!\n");
      _status = THREADFAIL;
   }
}

public static void main(String argv[]) {
   System.out.print("Entered the testcase\n");

   System.out.print("Create a thread\n");
   theThread t = new theThread();

   System.out.print("Start the thread\n");
   t.start();

   System.out.print("Wait for the thread to complete\n");
   try {
      t.join();
   }
   catch (InterruptedException e) {
      System.out.print("Join interrupted\n");
   }
   System.out.print("Check the thread status\n");
   if (t.status() != theThread.THREADPASS) {
      System.out.print("The thread failed\n");
   }

   System.out.print("Testcase completed\n");
   System.exit(0);
}

}
```

Canceling a thread

With the ability to end a thread externally, you can cancel threads that run long requests before they are completed on their own. You can use several APIs to cancel threads.

Some APIs provide well-defined points for the cancelation action to occur, or other mechanisms that allow one thread to control when another thread ends. Some APIs also provide a mechanism to run cleanup code before the thread ends, or to set the result of the canceled thread to a specific value.

Use thread cancelation carefully. If your API does not provide well-defined cancelation points or a mechanism for the thread to clean up the application data and locks, you might damage data or cause deadlocks within your application.

Note: In Java Development Kit version 1.2 or later, the resume, stop, and suspend methods for the thread class have been deprecated. This is because these methods are considered unsafe. The functions provided by these methods can be implemented by other mechanisms, such as checking the state of some variables. See the Sun Java tutorial for the suggested mechanism for a given version of Java.

   Related concepts

"Suspending a thread" on page 22

It is sometimes useful to stop a thread temporarily from processing in your application. When you suspend a thread, the state of the thread, including all the attributes and locks held by the thread, is maintained until that thread is resumed.

"Resuming a thread" on page 24

If you temporarily stopped a thread from processing in your application, you can resume the processing of the suspended thread at a later time at the same point that you suspended it.

Related information

➡ Sun Java tutorial

Example: Canceling a thread in a Pthread program

This example shows a Pthread program canceling a long-running thread.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
Filename: ATEST13.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST13
 Create/start a thread
 Wait a bit until we 'realize' the thread needs to be canceled
 Thread: Entered
 Thread: Looping or long running request
 Thread: Looping or long running request
 Thread: Looping or long running request
 Wait for the thread to complete, and release its resources
 Thread: Looping or long running request
 Thread status indicates it was canceled
 Main completed
*/
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define checkResults(string, val) {                \
 if (val) {                                        \
   printf("Failed with %d at %s", val, string); \
   exit(1);                                        \
 }                                                 \
}

void *theThread(void *parm)
{
   printf("Thread: Entered\n");
   while (1) {
      printf("Thread: Looping or long running request\n");
      pthread_testcancel();
      sleep(1);
   }
   return NULL;
}

int main(int argc, char **argv)
{
  pthread_t            thread;
  int                  rc=0;
  void                *status;

  printf("Enter Testcase - %s\n", argv[0]);

  printf("Create/start a thread\n");
  rc = pthread_create(&thread, NULL, theThread, NULL);
```

```
  checkResults("pthread_create()\n", rc);

  printf("Wait a bit until we 'realize' the thread needs to be canceled\n");
  sleep(3);
  rc = pthread_cancel(thread);
  checkResults("pthread_cancel()\n", rc);

  printf("Wait for the thread to complete, and release its resources\n");
  rc = pthread_join(thread, &status);
  checkResults("pthread_join()\n", rc);

  printf("Thread status indicates it was canceled\n");
  if (status != PTHREAD_CANCELED) {
     printf("Unexpected thread status\n");
  }

  printf("Main completed\n");
  return 0;
}
```

Example: Canceling a thread using a Java program

This example shows a Java program canceling a long-running thread.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
FileName: ATEST13.java

The output of this example is as follows:
 Entered the testcase
 Create a thread
 Start the thread
 Wait a bit until we 'realize' the thread needs to be canceled
 Thread: Entered
 Thread: Looping or long running request
 Thread: Looping or long running request
 Thread: Looping or long running request
 Wait for the thread to complete
 Thread status indicates it was canceled
 Testcase completed
*/
import java.lang.*;


public class ATEST13 {

  static class theThread extends Thread {
     public final static int THREADPASS     = 0;
     public final static int THREADFAIL     = 1;
     public final static int THREADCANCELED = 2;
     int        _status;

     public int status() {
        return _status;
     }
     public theThread() {
        _status = THREADFAIL;
     }
     public void run() {
        System.out.print("Thread: Entered\n");
        try {
           while (true) {
              System.out.print("Thread: Looping or long running request\n");
              try {
                 Thread.sleep(1000);
              }
```

```
            catch (InterruptedException e) {
                System.out.print("Thread: sleep interrupted\n");
            }
        }
    }
    catch (ThreadDeath d) {
        _status = THREADCANCELED;
    }
}
}

public static void main(String argv[]) {
    System.out.print("Entered the testcase\n");

    System.out.print("Create a thread\n");
    theThread t = new theThread();

    System.out.print("Start the thread\n");
    t.start();

    System.out.print("Wait a bit until we 'realize' the thread needs to be canceled\n");
    try {
        Thread.sleep(3000);
    }
    catch (InterruptedException e) {
        System.out.print("sleep interrupted\n");
    }
    t.stop();

    System.out.print("Wait for the thread to complete\n");
    try {
        t.join();
    }
    catch (InterruptedException e) {
        System.out.print("Join interrupted\n");
    }
    System.out.print("Thread status indicates it was canceled\n");
    if (t.status() != theThread.THREADCANCELED) {
        System.out.print("Unexpected thread status\n");
    }

    System.out.print("Testcase completed\n");
    System.exit(0);
}

}
```

Suspending a thread

It is sometimes useful to stop a thread temporarily from processing in your application. When you
suspend a thread, the state of the thread, including all the attributes and locks held by the thread, is
maintained until that thread is resumed.

Use thread suspension carefully. Suspending threads can easily cause application deadlocks and timeout
conditions. You can solve most problems that involve thread suspension by using other safer mechanisms
(such as synchronization primitives).

Note: The suspend method of the Java thread class has been deprecated.

Related concepts

"Synchronization techniques among threads" on page 33
When you create code that is threadsafe but still benefits from sharing data or resources between
threads, the most important aspect of programming becomes the ability to synchronize threads.

With the ability to end a thread externally, you can cancel threads that run long requests before they are completed on their own. You can use several APIs to cancel threads.

Example: Suspending a thread using a Java program

This example shows a Java program suspending an actively running thread.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
FileName: ATEST14.java

The output of this example is as follows:
 Entered the testcase
 Create a thread
 Start the thread
 Wait a bit until we 'realize' the thread needs to be suspended
 Thread: Entered
 Thread: Active processing
 Thread: Active processing
 Suspend the thread
 Wait a bit until we 'realize' the thread needs to be resumed
 Resume the thread
 Thread: Active processing
 Wait for the thread to complete
 Thread: Active processing
 Thread: Active processing
 Thread: Completed
 Testcase completed
*/
import java.lang.*;


public class ATEST14 {

    static class theThread extends Thread {
        public void run() {
            int        loop=6;
            System.out.print("Thread: Entered\n");
            while (--loop > 0) {
                System.out.print("Thread: Active processing\n");
                safeSleep(1000, "Thread: sleep interrupted\n");
            }
            System.out.print("Thread: Completed\n");
        }
    }

    public static void main(String argv[]) {
        System.out.print("Entered the testcase\n");

        System.out.print("Create a thread\n");
        theThread t = new theThread();

        System.out.print("Start the thread\n");
        t.start();

        System.out.print("Wait a bit until we 'realize' the thread needs to be suspended\n");
        safeSleep(2000, "Main first sleep interrupted\n");
        System.out.print("Suspend the thread\n");
        t.suspend();

        System.out.print("Wait a bit until we 'realize' the thread needs to be resumed\n");
        safeSleep(2000, "Main second sleep interrupted\n");
        System.out.print("Resume the thread\n");
        t.resume();
```

```
        System.out.print("Wait for the thread to complete\n");
        try {
            t.join();
        }
        catch (InterruptedException e) {
            System.out.print("Join interrupted\n");
        }

        System.out.print("Testcase completed\n");
        System.exit(0);
    }

    public static void safeSleep(long milliseconds, String s) {
        try {
            Thread.sleep(milliseconds);
        }
        catch (InterruptedException e) {
            System.out.print(s);
        }
    }

}
```

Resuming a thread

If you temporarily stopped a thread from processing in your application, you can resume the processing of the suspended thread at a later time at the same point that you suspended it.

The resumed thread holds the same locks and has the same attributes that it had when you suspended it.

Note: The suspend method of the Java thread class has been deprecated.

> **Related concepts**
>
> "Canceling a thread" on page 19
> With the ability to end a thread externally, you can cancel threads that run long requests before they are completed on their own. You can use several APIs to cancel threads.

Waiting for a thread to end

When you use threads, it is important to know when a thread finishes processing. Waiting for a thread to perform an action or for an event to happen is called thread synchronization.

It is typically sufficient to wait for a thread to end. When the thread ends, the application is alerted that the work that the thread was assigned is completed or that the thread has failed. If a status is set by the thread and is supported by your specific API, you can use the status to determine whether the thread successfully completed its work.

When a thread ends, the system reclaims its resources for reuse. You can also wait for a thread to end to reclaim more of the thread's resources for use by other threads.

Waiting for a group of threads to end can also be good for a larger application, such as a server. The large application can then give large amounts of application work to "worker" threads and have a single "boss" thread coordinating the work of the subordinate threads. Your API might directly support this kind of waiting.

Example: Waiting for threads in a Pthread program

This example shows a Pthread program starting several threads, waiting for them to finish, and checking their status after they are completed.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
Filename: ATEST13.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST15
 Create/start some worker threads
 Thread 00000000 000001a4: Entered
 Thread 00000000 000001a4: Working
 Wait for worker threads to complete, release their resources
 Thread 00000000 000001a8: Entered
 Thread 00000000 000001a8: Working
 Thread 00000000 000001a5: Entered
 Thread 00000000 000001a5: Working
 Thread 00000000 000001a6: Entered
 Thread 00000000 000001a6: Working
 Thread 00000000 000001a7: Entered
 Thread 00000000 000001a7: Working
 Thread 00000000 000001a4: Done with work
 Thread 00000000 000001a8: Done with work
 Thread 00000000 000001a6: Done with work
 Thread 00000000 000001a7: Done with work
 Thread 00000000 000001a5: Done with work
 Check all thread's results
 Main completed
*/
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define THREADGROUPSIZE   5

#define checkResults(string, val) {              \
 if (val) {                                      \
   printf("Failed with %d at %s", val, string);  \
   exit(1);                                       \
 }                                               \
}

void *theThread(void *parm)
{
   printf("Thread %.8x %.8x: Entered\n", pthread_getthreadid_np());
   printf("Thread %.8x %.8x: Working\n", pthread_getthreadid_np());
   sleep(15);
   printf("Thread %.8x %.8x: Done with work\n", pthread_getthreadid_np());
   return NULL;
}

int main(int argc, char **argv)
{
  pthread_t            thread[THREADGROUPSIZE];
  void                 *status[THREADGROUPSIZE];
  int                  i;
  int                  rc=0;

  printf("Enter Testcase - %s\n", argv[0]);

  printf("Create/start some worker threads\n");
  for (i=0; i <THREADGROUPSIZE; ++i)
 {
   rc = pthread_create(&thread[i], NULL, theThread, NULL);
   checkResults("pthread_create()\n", rc);
 }

  printf("Wait for worker threads to complete, release their resources\n");
```

```
    for (i=0; i <THREADGROUPSIZE; ++i) {
rc = pthread_join(thread[i], &status[i]);
    checkResults("pthread_join()\n", rc);
  }

  printf("Check all thread's results\n");
  for (i=0; i <THREADGROUPSIZE; ++i) {
if (status[i] != NULL) {
      printf("Unexpected thread status\n");
    }
  }

  printf("Main completed\n");
  return 0;
}
```

Example: Waiting for threads using a Java program

This example shows a Java program starting several threads, waiting for them to finish, and checking their status after completion.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
FileName: ATEST15.java

The output of this example is as follows:
 Entered the testcase
 Create some worker threads
 Start the thread
 Thread Thread-1: Entered
 Thread Thread-1: Working
 Thread Thread-2: Entered
 Thread Thread-2: Working
 Thread Thread-3: Entered
 Thread Thread-3: Working
 Thread Thread-4: Entered
 Thread Thread-4: Working
 Wait for worker threads to complete
 Thread Thread-5: Entered
 Thread Thread-5: Working
 Thread Thread-1: Done with work
 Thread Thread-2: Done with work
 Thread Thread-3: Done with work
 Thread Thread-4: Done with work
 Thread Thread-5: Done with work
 Check all thread's results
 Testcase completed
*/
import java.lang.*;


public class ATEST15 {
   public final static int THREADGROUPSIZE   = 5;

   static class theThread extends Thread {
      public final static int THREADPASS     = 0;
      public final static int THREADFAIL     = 1;
      int         _status;

      public int status() {
         return _status;
      }
      public theThread() {
         _status = THREADFAIL;
      }
      public void run() {
```

```
            System.out.print("Thread " + getName() + ": Entered\n");
            System.out.print("Thread " + getName() + ": Working\n");
            safeSleep(15000, "Thread " + getName() + " work");
            System.out.print("Thread " + getName() + ": Done with work\n");
            _status = THREADPASS;
        }
    }

    public static void main(String argv[]) {
        int        i=0;
        theThread    thread[] = new theThread[THREADGROUPSIZE];

        System.out.print("Entered the testcase\n");

        System.out.print("Create some worker threads\n");
        System.out.print("Start the thread\n");
        /* We won't use a ThreadGroup for this example, because we'd */
        /* still have to join all the threads individually          */
        for (i=0; i <THREADGROUPSIZE; ++i) {
    thread[i] = new theThread();
            thread[i].start();
        }

        System.out.print("Wait for worker threads to complete\n");
        for (i=0; i <THREADGROUPSIZE; ++i) {
try {
                thread[i].join();
        }
        catch (InterruptedException e) {
            System.out.print("Join interrupted\n");
        }
    }
        System.out.print("Check all thread's results\n");
        for (i=0; i <THREADGROUPSIZE; ++1) {
if (thread[i].status() != theThread.THREADPASS) {
            System.out.print("Unexpected thread status\n");
        }
    }

        System.out.print("Testcase completed\n");
        System.exit(0);
    }

    public static void safeSleep(long milliseconds, String s) {
        try {
            Thread.sleep(milliseconds);
        }
        catch (InterruptedException e) {
            System.out.print(s);
        }
    }
}
```

Yielding the processor to another thread

At times, your application can benefit if a thread yields the processor to another thread in the system.

When a thread yields the processor, the system immediately allows another active thread of equal or higher priority to run. If no threads of equal or higher priority are currently ready to run, yielding the processor has no effect. Beyond the immediate action, yielding the processor does not have any additional predictable behavior related to the scheduling of the threads in the system.

The System i platform provides a fully preemptive multitasking scheduling algorithm. A thread that exists in a well-written application should seldom need to yield the processor to other threads. This is because more predictable application programming interfaces (APIs) for synchronizing threads are available.

Thread safety

A function is threadsafe if you can start it simultaneously in multiple threads within the same process. A function is threadsafe only if all the functions it calls are also threadsafe.

Related concepts

"Function calls that are not threadsafe" on page 54
Multithreaded application at times requires access to functions or system services that are not threadsafe. There are few completely safe alternatives for calling these functions.

Related reference

"Activation groups and threads" on page 5
All programs and service programs are activated within a substructure of a job that is called an activation group. This substructure contains the resources necessary to run the programs, including static and heap storage, and temporary data management resources.

"Existing application evaluation" on page 32
When writing a multithreaded application, you must evaluate all the parts of the application and all the services that the application uses for thread safety.

Storage usage and threaded applications

When you write a threaded application, it is important to understand the visibility and scope of various classes of storage.

When your application declares variables, multiple threads might be able to access or use these variables. The visibility and scope of the storage that are used for the variables of the application affect the function of the application.

- Global storage

  Global storage that you declare in one of your application source files is visible to all other source files or modules in your application. All threads that run code in your application share the same global storage. This unintended sharing of storage is a common safety problem.

- Static storage

  Static storage is global storage with visibility that is restricted to the source file, module, or function in which the storage or variable was declared. All threads that run code from that module share the same static storage.

- Thread local storage

  A thread local storage variable is visible to all source files in your application, a single source module, or a single function, depending on how the variable is declared. Thread local storage is not visible to other threads in the process. When a thread is started, copies of thread local storage variables are created for the thread.

- Heap storage

  Heap storage is allocated and deallocated dynamically by your application (for example, by malloc() and free() in C, or new and delete in Java or C++). All threads that run code in your application share the same heap.

  If you give another thread a pointer to the heap storage that has been allocated (by storing the pointer in static or global storage, or by otherwise passing the pointer to another thread), that thread can use or deallocate the storage. This unintended sharing of storage is a common thread safety problem.

- Automatic storage

Automatic or local storage is allocated automatically by your language when you declare a variable that is private to a function, method, or subroutine. Local storage is not visible to other threads in the process. Each time those threads call the function, method, or subroutine, the thread allocates new versions of the automatic variables. Each thread gets its own automatic storage. A thread should not access automatic storage from another thread because of the complex serialization and synchronization issues involved.

The operating system further restricts the scope of global, static, thread local, and heap storage to the activation group that your application code is running in. This means that application code or threads, which are running in different activation groups but are using the same global or static variables, access different versions of those variables and their storage. Similarly, heap storage allocated in one activation group might not be deallocated in another activation group, although the storage can be used by different activation groups.

Some languages support only global or static storage classes for variables. These languages are not easy to use in threaded applications. You should not use threads in these languages.

Job-scoped resources and thread safety

Many system and application resources are available only within a defined job. When writing a multithreaded application or service, you must evaluate your use of job-scoped resources. The use of these resources must not conflict with or negatively affect other threads in the process.

Some resources (open database files, for example) are scoped to the activation group. For the purposes of this discussion, resources that are scoped to the activation group must be treated with the same considerations as job-scoped resources. All of the threads that use the activation group use these resources.

Your application can use appropriate synchronization techniques if threads in your application modify these resources when other threads are using them. You need to ensure a consistent view of the resources between your threads.

Here are some of the common job-scoped or activation-group-scoped resources:
* Heap, static, and global storage

  The most commonly shared resource. For more information, see "Storage usage and threaded applications" on page 28.
* Open files

  After you open a file, threads in a process can directly share integrated file system files and database files by passing the file handle pointer or file descriptor number to another thread. The working directory is always scoped to the process.
* Locales

  The locale of an application is an activation group resource. All threads share the locale. Changing the locale affects other threads with regard to collating sequences or other locale information.
* Coded character set identifier

  Changing the coded character set identifier (CCSID) of a job affects the current data translation or representation of all threads in that job.
* Environment variables

  Your application commonly uses environment variables for configuration and optional behavior. All the threads in the job share these environment variables.

API thread safety classifications

Each API has a threadsafe classification. Before using an API, you need to determine if it is safe to call that API in your multithreaded program.

There are several types of threadsafe classifications.

- Threadsafe: yes

  This classification indicates that you can safely call the API simultaneously in multiple threads without restrictions. This classification also indicates that all functions called by this API are threadsafe.

- Threadsafe: conditional

  This classification indicates that not all functions provided by the API are threadsafe. The Usage Notes section of the API provides information that relates to thread safety limitations. Many APIs are classified as conditionally threadsafe because either some underlying system support is not threadsafe or the API can call an exit point. For example, many of the file system APIs are completely threadsafe when used with files in a threadsafe file system. Some conditionally threadsafe APIs might deny access under some circumstances. The API usage notes describe the conditions that cause the function to deny access.

- Threadsafe: no

  This classification indicates that the API is not threadsafe and that the API must not be used in a multithreaded program. Although some APIs that are not threadsafe might deny access, most APIs that are not threadsafe do not. Unlike in CL commands, no diagnostic message is in the job log to indicate that an API that is not threadsafe has been called (other than APIs that deny access). There are techniques for calling functions that are not threadsafe in multithreaded programs.

  Related concepts

  Application programming interfaces

CL commands and thread safety

The Integrated Language Environment (ILE) CL run time and compiler-generated code are threadsafe. Original program model (OPM) CL programs are not threadsafe.

For any ILE CL code that is compiled before Version 4 Release 3, or for any OPM CL code, the CL run time sends a CPD000B diagnostic message and continues to run with unpredictable results. This might or might not be threadsafe, depending on the underlying code.

Command analyzer is threadsafe. For a given command, if the threadsafe attribute (THDSAFE) is *NO and the multithreaded job action attribute (MLTTHDACN) is set to *NORUN or *MSG, the command analyzer does one of the following operations when such a command is called in a job capable of multithreaded operations:

- If MLTTHDACN is set to *NORUN in a multithreaded job, command analyzer sends a CPD000D diagnostic message, and the command does not run. A CPF0001 escape message then follows the CPD000D diagnostic message.
- If MLTTHDACN is set to *MSG in a multithreaded job, command analyzer sends a CPD000D diagnostic message, and the program continues to run with unpredictable results.
- If the job is capable of running multiple threads but is not actually multithreaded, command analyzer allows commands that are not threadsafe to run without interference.

If MLTTHDACN is set to *RUN, the command analyzer does not send a diagnostic message and allows the command to run. The results of this are unpredictable. Also, MLTTHDACN only applies to commands whose THDSAFE value is *NO. To determine the THDSAFE and MLTTHDACN values for a command, use the Display Command (DSPCMD) command.

The MLTTHDACN value of some commands is set to *SYSVAL. In this case, command analyzer uses the QMLTTHDACN system value to decide how to process the command.

To display the setting of this value on the i5/OS operating system, use the following command:

DSPSYSVAL SYSVAL(QMLTTHDACN)

To change the setting of this value on i5/OS, use the following command:

CHGSYSVAL SYSVAL(QMLTTHDACN) VALUE(x)

Sample output from DSPSYSVAL:

```
Display System Value

System value . . . . . . : QMLTTHDACN
Description . . . . . . : Multithreaded job action


Multithreaded job
action . . . . . . . : 2
1=Perform the function that is not
threadsafe without sending a message
2=Perform the function that is not
threadsafe and send an informational
message
3=Do not perform the function that is
not threadsafe
```

> **Related concepts**
>
> "Multithreaded programming techniques" on page 32
> If you are going to write multithreaded applications or system applications for the i5/OS operating system, you need to be familiar with several techniques.
>
> "Function calls that are not threadsafe" on page 54
> Multithreaded application at times requires access to functions or system services that are not threadsafe. There are few completely safe alternatives for calling these functions.

Denied access functions and thread safety

To prevent problems in system integrity and to prevent data damages, certain APIs and commands are either conditionally threadsafe or not threadsafe. These APIs and commands deny all or partial access.

The criteria for denying access are as follows:

- Multithread capability

  In this case, the decision to deny access to a function is made by checking the multithread-capable job attribute. If the job is capable of supporting threads, regardless of the current number of threads in the job, you cannot call the function. When a function denies access based on multithread capability, it sends a CPF1892 (Function &1 not allowed) escape message to the caller. You must call functions that deny access based on multithread capability in separate jobs.

- Initial thread

  Some functions deny access based on whether the initial thread of the job calls the function. If you call the function in a secondary thread, the function denies access by sending a CPF180C (Function &1 not allowed) escape message to the caller. You can call a function that denies access from a secondary thread by routing a request to the initial thread and having the function called in the initial thread. Overriding a database file is an example of a function that is only allowed in the initial thread.

- More than one thread

  In this case, the number of threads in the job causes the decision to deny access. If more than one thread is in the job, the function sends a CPF180B (Function &1 not allowed) escape message to the caller. Other functions that return error numbers sets errno to ENOTSAFE.

  You can call a function that denies access when there are multiple threads by restricting its use to when only one thread is active. An example of such a function is any file system API that is used to access a file in a file system that is not threadsafe.

  > **Related information**
  >
  > Reclaim Resources (RCLRCS) command

Exit points

Using the i5/OS registration facility, you can define exit points for functions in an application and register programs that run at those exit points.

Some i5/OS services also support the registration facility for registering exit programs. These i5/OS services have predefined exit points that are registered when those services are installed. The registration facility itself is threadsafe. You can use the registration facility to specify attributes of thread safety and multithreaded job actions for exit program entries.

However, do not consider the existing exit programs to be threadsafe without careful evaluation, even though you can call the exit programs in a multithreaded job. The same restrictions apply to exit programs as to any other code that runs in a multithreaded job. For example, only exit programs written using a threadsafe Integrated Language Environment (ILE) language can be made threadsafe.

Related reference

"Existing application evaluation"
When writing a multithreaded application, you must evaluate all the parts of the application and all the services that the application uses for thread safety.

"Database considerations for multithreaded programming" on page 6
When you use databases in multithreaded programs, be aware of these considerations.

Multithreaded programming techniques

If you are going to write multithreaded applications or system applications for the i5/OS operating system, you need to be familiar with several techniques.

The concepts that are described in this section pertain to all programming languages. To determine how each language enables these concepts, refer to the Language topic in the information center for the specific language.

Related reference

"CL commands and thread safety" on page 30
The Integrated Language Environment (ILE) CL run time and compiler-generated code are threadsafe. Original program model (OPM) CL programs are not threadsafe.

Existing application evaluation

When writing a multithreaded application, you must evaluate all the parts of the application and all the services that the application uses for thread safety.

How each service or API that is used by the application uses its storage resources is an important consideration for providing threadsafe applications. If you do not use storage in a threadsafe manner, data in your application is likely to be damaged.

Other critical aspects in creating threadsafe applications are the APIs and system services on which your application or your application services rely. To be threadsafe, your storage usage, and all APIs and services that you use directly or indirectly must be threadsafe. APIs and system resources that you use indirectly are particularly difficult to evaluate. See the APIs topic for the specific APIs or system services you use to see if they are threadsafe.

You will probably call system services, APIs, or other applications for which you do not have the source code. For many of these services, the threadsafe classification is not documented and you must assume that these services are not threadsafe.

Related concepts

"Thread safety" on page 28
A function is threadsafe if you can start it simultaneously in multiple threads within the same process.
A function is threadsafe only if all the functions it calls are also threadsafe.

Related reference

"Exit points" on page 32
Using the i5/OS registration facility, you can define exit points for functions in an application and
register programs that run at those exit points.

Synchronization techniques among threads

When you create code that is threadsafe but still benefits from sharing data or resources between threads,
the most important aspect of programming becomes the ability to synchronize threads.

Synchronization is the cooperative act of two or more threads that ensures that each thread reaches a
known point of operation in relationship to other threads before continuing. Attempting to share
resources without correctly using synchronization is the most common cause of damage to application
data.

Typically, synchronizing two threads involves the use of one or more synchronization primitives.
Synchronization primitives are low-level functions or application objects (not i5/OS objects) that your
application uses or creates to provide the synchronization behavior that the application requires.

Here are the most common synchronization primitives in order of least to most computationally
expensive:
- Compare and swap
- Mutual exclusion (mutexes) and threads
- Semaphores and threads
- Condition variables and threads
- Threads as synchronization primitives
- Space location locks
- Object locks

These concepts pertain to all programming languages. To determine how each language enables these
concepts, refer to the Language topic in the information center for the specific language.

Related concepts

"Suspending a thread" on page 22
It is sometimes useful to stop a thread temporarily from processing in your application. When you
suspend a thread, the state of the thread, including all the attributes and locks held by the thread, is
maintained until that thread is resumed.

"Function calls that are not threadsafe" on page 54
Multithreaded application at times requires access to functions or system services that are not
threadsafe. There are few completely safe alternatives for calling these functions.

The Compare and Swap instruction

You can use the Machine Interface's (MI) Compare and Swap (CMPSWP) instruction to access data in a
multithreaded program.

The CMPSWP instruction compares the value of a first compare operand to the value of a second
compare operand. If the two values are equal, the swap operand is stored in the location of the second
compare operand. If the two values are unequal, the second compare operand is stored into the location
of the first compare operand.

When an equal comparison occurs, the CMPSWP instruction assures that no access by another CMPSWP instruction will occur at the location of the second compare operand between the moment that the second compare operand is fetched for comparison and the moment that the swap operand is stored at the location of the second compare operand.

When an unequal comparison occurs, no atomicity guarantees are made regarding the store to the first compare operand location and other CMPSWP instruction access. Thus only the second compare operand should be a variable shared for concurrent processing control.

The following code example for a C macro can be used to atomically increment or decrement an integer variable.

```
#ifndef __cmpswp_h
    #include <mih/cmpswp.h>
#endif

#define ATOMICADD ( var, val, rc ) {              \
    int aatemp1 = (var);                          \
    int aatemp2  = aatemp1 + val;                 \
    while( ! _CMPSWP( &aatemp1, &var, aatemp2 ) ) \
        aatemp2 = aatemp1 + val;                  \
    rc = aatemp2;                                 \
}
```

In the code example, *var* is an integer to be incremented or decremented, *val* is an integer value to be added or subtracted from *var*, and *rc* is the resultant value.

Mutexes and threads

A mutual exclusion (mutex) is used cooperatively between threads to ensure that only one of the cooperating threads is allowed to access the data or run certain application code at a time.

The word mutex is shorthand for a primitive object that provides MUTual EXclusion between threads. For the purposes of this introduction, you can think of mutexes as similar to critical sections and monitors.

The mutex is typically logically associated with the data it protects by the application. For example, PAYROLL DATA has a PAYROLL MUTEX associated with it. The application code always locks the PAYROLL MUTEX before accessing the PAYROLL DATA. The mutex prevents access to the data by a thread only if that thread uses the mutex before accessing the data.

Create, lock, unlock, and destroy are operations typically performed on a mutex. Any thread that successfully locks the mutex becomes the owner of the mutex until it unlocks the mutex. Any thread that attempts to lock the mutex later has to wait until the owner unlocks the mutex. When the owner unlocks the mutex, control is returned to a waiting thread with that thread becoming the owner of the mutex. There can be only one owner of a mutex at a time.

Mutex wait operations can be recursive. A recursive mutex allows the owner thread to lock the mutex repeatedly. The owner of the mutex remains the same until the number of unlock requests equals the number of successful lock requests. Mutex waits can time out after a user-specified amount of time, or return immediately if they cannot acquire the lock. For more information, see your API documentation about mutex primitives available for your application.

Example: Using mutexes in Pthread programs:

This example shows a Pthread program starting several threads that protect access to shared data with a mutual exclusion (mutex).

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
Filename: ATEST16.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST16
 Hold Mutex to prevent access to shared data
 Create/start threads
 Wait a bit until we are 'done' with the shared data
 Thread 00000000 00000019: Entered
 Thread 00000000 0000001a: Entered
 Thread 00000000 0000001b: Entered
 Unlock shared data
 Wait for the threads to complete, and release their resources
 Thread 00000000 00000019: Start critical section, holding lock
 Thread 00000000 00000019: End critical section, release lock
 Thread 00000000 0000001a: Start critical section, holding lock
 Thread 00000000 0000001a: End critical section, release lock
 Thread 00000000 0000001b: Start critical section, holding lock
 Thread 00000000 0000001b: End critical section, release lock
 Clean up
 Main completed
*/
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define checkResults(string, val) {               \
 if (val) {                                        \
   printf("Failed with %d at %s", val, string); \
   exit(1);                                        \
 }                                                 \
}

#define               NUMTHREADS    3
pthread_mutex_t        mutex = PTHREAD_MUTEX_INITIALIZER;
int                    sharedData=0;
int                    sharedData2=0;

void *theThread(void *parm)
{
   int   rc;
   printf("Thread %.8x %.8x: Entered\n", pthread_getthreadid_np());
   rc = pthread_mutex_lock(&mutex);
   checkResults("pthread_mutex_lock()\n", rc);
   /********** Critical Section *******************/
   printf("Thread %.8x %.8x: Start critical section, holding lock\n",
          pthread_getthreadid_np());
   /* Access to shared data goes here */
   ++sharedData; --sharedData2;
   printf("Thread %.8x %.8x: End critical section, release lock\n",
          pthread_getthreadid_np());
   /********** Critical Section *******************/
   rc = pthread_mutex_unlock(&mutex);
   checkResults("pthread_mutex_unlock()\n", rc);
   return NULL;
}

int main(int argc, char **argv)
{
  pthread_t             thread[NUMTHREADS];
  int                   rc=0;
  int                   i;

  printf("Enter Testcase - %s\n", argv[0]);

  printf("Hold Mutex to prevent access to shared data\n");
  rc = pthread_mutex_lock(&mutex);
```

```
    checkResults("pthread_mutex_lock()\n", rc);

  printf("Create/start threads\n");
  for (i=0; i<NUMTHREADS; ++i) {
 rc = pthread_create(&thread[i], NULL, theThread, NULL);
     checkResults("pthread_create()\n", rc);
  }

  printf("Wait a bit until we are 'done' with the shared data\n");
  sleep(3);
  printf("Unlock shared data\n");
  rc = pthread_mutex_unlock(&mutex);
  checkResults("pthread_mutex_lock()\n",rc);

  printf("Wait for the threads to complete, and release their resources\n");
  for (i=0; i <NUMTHREADS; ++i) {
   rc = pthread_join(thread[i], NULL);
     checkResults("pthread_join()\n", rc);
  }

  printf("Clean up\n");
  rc = pthread_mutex_destroy(&mutex);
  printf("Main completed\n");
  return 0;
}
```

Example: Using mutexes in a Java program:

This example shows a Java program creating a critical section of code. In Java, any object or array can function similarly to a mutual exclusion (mutex) using the synchronized keyword on a block of code or a method.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
FileName: ATEST16.java
The output of this example is as follows:
 Entered the testcase
 Synchronize to prevent access to shared data
 Create/start the thread
 Thread Thread-1: Entered
 Thread Thread-2: Entered
 Wait a bit until we're 'done' with the shared data
 Thread Thread-3: Entered
 Unlock shared data
 Thread Thread-1: Start critical section, in synchronized block
 Thread Thread-1: End critical section, leave synchronized block
 Thread Thread-2: Start critical section, in synchronized block
 Thread Thread-2: End critical section, leave synchronized block
 Thread Thread-3: Start critical section, in synchronized block
 Thread Thread-3: End critical section, leave synchronized block
 Wait for the threads to complete
 Testcase completed
*/
import java.lang.*;


public class ATEST16 {
   public final static int NUMTHREADS   = 3;
   public static int sharedData  = 0;
   public static int sharedData2 = 0;
   /* Any real java object or array would suit for synchronization      */
   /* We invent one here since we have two unique data items to synchronize */
   /* An in this simple example, they're not in an object               */
   static class theLock extends Object {
```

```
        }
    static public theLock lockObject = new theLock();

    static class theThread extends Thread {
        public void run() {
            System.out.print("Thread " + getName() + ": Entered\n");
            synchronized (lockObject) {
                /********** Critical Section ******************/
                System.out.print("Thread " + getName() +
                                    ": Start critical section, in synchronized block\n");
                ++sharedData; --sharedData2;
                System.out.print("Thread " + getName() +
                                    ": End critical section, leave synchronized block\n");
                /********** Critical Section ******************/
            }
        }
    }

    public static void main(String argv[]) {
        theThread threads[] = new theThread[NUMTHREADS];
        System.out.print("Entered the testcase\n");

        System.out.print("Synchronize to prevent access to shared data\n");
        synchronized (lockObject) {

            System.out.print("Create/start the thread\n");
            for (int i=0; i<NUMTHREADS; ++i) {
            threads[i] = new theThread();
                threads[i].start();
            }

            System.out.print("Wait a bit until we're 'done' with the shared data\n");
            try {
                Thread.sleep(3000);
            }
            catch (InterruptedException e) {
                System.out.print("sleep interrupted\n");
            }
            System.out.print("Unlock shared data\n");
        }

        System.out.print("Wait for the threads to complete\n");
        for(int i=0; i <NUMTHREADS; ++i) {
        threads[i].join();
            }
            catch (InterruptedException e) {
                System.out.print("Join interrupted\n");
            }
        }

        System.out.print("Testcase completed\n");
        System.exit(0);
    }

}
```

Semaphores and threads

Semaphores (sometimes referred to as counting semaphores) can be used to control access to shared
resources. A semaphore can be thought of as an intelligent counter. Every semaphore has a current count,
which is greater than or equal to 0.

Any thread can decrement the count to lock the semaphore (this is also called waiting on the semaphore).
Attempting to decrement the count past 0 causes the thread that is calling to wait for another thread to
unlock the semaphore.

Any thread can increment the count to unlock the semaphore (this is also called posting the semaphore). Posting a semaphore might wake up a waiting thread if there is one present.

In their simplest form (with an initial count of 1), semaphores can be thought of as a mutual exclusion (mutex). The important distinction between semaphores and mutexes is the concept of ownership. No ownership is associated with a semaphore. Unlike mutexes, it is possible for a thread that never waited for (locked) the semaphore to post (unlock) the semaphore. This can cause unpredictable application behavior. You must avoid this if possible.

The operating system provides the following additional capabilities of some semaphore APIs.
- More complete management capabilities, including permissions on semaphores that are similar to file permissions
- The ability to group semaphores in sets and perform atomic operations on the group
- The ability to do multicount wait and post operations
- The ability to wait for a semaphore to have a count of 0
- The ability to undo operations that were done by another thread under certain conditions

Note: Java does not have the ability to use semaphores.

Example: Using semaphores in Pthread programs to protect shared data:

This example shows a Pthread program starting several threads that protect access to shared data with a semaphore set.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
Filename: ATEST17.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST17
 Wait on semaphore to prevent access to shared data
 Create/start threads
 Wait a bit until we are 'done' with the shared data
 Thread 00000000 00000020: Entered
 Thread 00000000 0000001f: Entered
 Thread 00000000 0000001e: Entered
 Unlock shared data
 Wait for the threads to complete, and release their resources
 Thread 00000000 0000001f: Start critical section, holding semaphore
 Thread 00000000 0000001f: End critical section, release semaphore
 Thread 00000000 00000020: Start critical section, holding semaphore
 Thread 00000000 00000020: End critical section, release semaphore
 Thread 00000000 0000001e: Start critical section, holding semaphore
 Thread 00000000 0000001e: End critical section, release semaphore
 Clean up
 Main completed
*/
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>

#define checkResults(string, val) {                 \
 if (val) {                                         \
   printf("Failed with %d at %s", val, string); \
```

```
     exit(1);                                                   \
   }                                                            \
 }

#define                NUMTHREADS    3
int                    sharedData=0;
int                    sharedData2=0;

/* Simple semaphore used here is actually a set of 1                 */
int                    semaphoreId=-1;
/* Simple lock operation. 0=which-semaphore, -1=decrement, 0=noflags  */
struct sembuf lockOperation = { 0, -1, 0};
/* Simple unlock operation. 0=which-semaphore, 1=increment, 0=noflags */
struct sembuf unlockOperation = { 0, 1, 0};

void *theThread(void *parm)
{
   int   rc;
   printf("Thread %.8x %.8x: Entered\n", pthread_getthreadid_np());
   rc = semop(semaphoreId, &lockOperation, 1);
   checkResults("semop(lock)\n",rc);
   /********** Critical Section ********************/
   printf("Thread %.8x %.8x: Start critical section, holding semaphore\n",
          pthread_getthreadid_np());
   /* Access to shared data goes here */
   ++sharedData; --sharedData2;
   printf("Thread %.8x %.8x: End critical section, release semaphore\n",
          pthread_getthreadid_np());
   /********** Critical Section ********************/
   rc = semop(semaphoreId, &unlockOperation, 1);
   checkResults("semop(unlock)\n",rc);
   return NULL;
}

int main(int argc, char **argv)
{
  pthread_t            thread[NUMTHREADS];
  int                  rc=0;
  int                  i;


  printf("Enter Testcase - %s\n", argv[0]);
  /* Create a private semaphore set with 1 semaphore that only I can use */
  semaphoreId = semget(IPC_PRIVATE, 1, S_IRUSR|S_IWUSR);
  if (semaphoreId &lt 0) { printf("semget failed, err=%d\n",errno); exit(1); }
  /* Set the semaphore (#0 in the set) count to 1. Simulate a mutex */
  rc = semctl(semaphoreId, 0, SETVAL, (int)1);
  checkResults("semctl(SETALL)\n", rc);

  printf("Wait on semaphore to prevent access to shared data\n");
  rc = semop(semaphoreId, &lockOperation, 1);
  checkResults("semop(lock)\n",rc);

  printf("Create/start threads\n");
  for (i=0; i <NUMTHREADS; ++i) {
  checkResults("pthread_create()\n", rc);
  }

  printf("Wait a bit until we are 'done' with the shared data\n");
  sleep(3);
  printf("Unlock shared data\n");
  rc = semop(semaphoreId, &unlockOperation, 1);
  checkResults("semop(unlock)\n",rc);


  printf("Wait for the threads to complete, and release their resources\n");
  for (i=0; i <NUMTHREADS; ++i) {
```

```
rc = pthread_join(thread[i], NULL);
    checkResults("pthread_join()\n", rc);
}

printf("Clean up\n");
rc = semctl(semaphoreId, 0, IPC_RMID);
checkResults("semctl(removeID)\n", rc);
printf("Main completed\n");
return 0;
}
```

Condition variables and threads

Condition variables allow threads to wait for certain events or conditions to occur and they notify other threads that are also waiting for the same events or conditions.

The thread can wait on a condition variable and broadcast a condition such that one or all of the threads that are waiting on the condition variable become active. You can consider condition variables to be similar to using events to synchronize threads on other platforms.

If a thread sends a signal to wake up all threads waiting on a condition variable and there are no threads waiting on that condition variable, the signal is discarded and no action is taken.

Note: It is possible for one thread to signal a condition immediately before another thread begins waiting on that condition variable, resulting in no action.

Locking protocols that use mutual exclusions (mutexes) are typically used with condition variables. If you use locking protocols, your application can ensure that a thread does not lose a signal that was intended to wake the thread up.

Example: Using condition variables in Pthread programs:

This example shows a Pthread program using condition variables to notify threads of a condition. Notice what mutual exclusion (mutex) locking protocol is used.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
Filename: ATEST18.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST18
 Create/start threads
 Producer: 'Finding' data
 Consumer Thread 00000000 00000022: Entered
 Consumer Thread 00000000 00000023: Entered
 Consumer Thread 00000000 00000022: Wait for data to be produced
 Consumer Thread 00000000 00000023: Wait for data to be produced
 Producer: Make data shared and notify consumer
 Producer: Unlock shared data and flag
 Producer: 'Finding' data
 Consumer Thread 00000000 00000022: Found data or Notified, CONSUME IT while holding lock
 Consumer Thread 00000000 00000022: Wait for data to be produced
 Producer: Make data shared and notify consumer
 Producer: Unlock shared data and flag
 Producer: 'Finding' data
 Consumer Thread 00000000 00000023: Found data or Notified, CONSUME IT while holding lock
 Consumer Thread 00000000 00000023: Wait for data to be produced
 Producer: Make data shared and notify consumer
 Producer: Unlock shared data and flag
 Producer: 'Finding' data
 Consumer Thread 00000000 00000022: Found data or Notified, CONSUME IT while holding lock
 Consumer Thread 00000000 00000022: All done
 Producer: Make data shared and notify consumer
```

```
 Producer: Unlock shared data and flag
 Wait for the threads to complete, and release their resources
 Consumer Thread 00000000 00000023: Found data or Notified, CONSUME IT while holding lock
 Consumer Thread 00000000 00000023: All done
 Clean up
 Main completed
*/
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define checkResults(string, val) {                \
 if (val) {                                        \
   printf("Failed with %d at %s", val, string);    \
   exit(1);                                         \
 }                                                 \
}


#define               NUMTHREADS    2
pthread_mutex_t        dataMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t         dataPresentCondition = PTHREAD_COND_INITIALIZER;
int                    dataPresent=0;
int                    sharedData=0;

void *theThread(void *parm)
{
   int   rc;
   int   retries=2;

   printf("Consumer Thread %.8x %.8x: Entered\n", pthread_getthreadid_np());
   rc = pthread_mutex_lock(&dataMutex);
   checkResults("pthread_mutex_lock()\n", rc);

   while (retries--) {
      /* The boolean dataPresent value is required for safe use of */
      /* condition variables. If no data is present we wait, other */
      /* wise we process immediately.                              */
      while (!dataPresent) {
         printf("Consumer Thread %.8x %.8x: Wait for data to be produced\n");
         rc = pthread_cond_wait(&dataPresentCondition, &dataMutex);
         if (rc) {
            printf("Consumer Thread %.8x %.8x: condwait failed, rc=%d\n",rc);
            pthread_mutex_unlock(&dataMutex);
            exit(1);
         }
      }
      printf("Consumer Thread %.8x %.8x: Found data or Notified, "
             "CONSUME IT while holding lock\n",
             pthread_getthreadid_np());
      /* Typically an application should remove the data from being    */
      /* in the shared structure or Queue, then unlock. Processing      */
      /* of the data does not necessarily require that the lock is held */
      /* Access to shared data goes here */
      --sharedData;
      /* We consumed the last of the data */
      if (sharedData==0) {dataPresent=0;}
      /* Repeat holding the lock. pthread_cond_wait releases it atomically */
   }
   printf("Consumer Thread %.8x %.8x: All done\n",pthread_getthreadid_np());
   rc = pthread_mutex_unlock(&dataMutex);
   checkResults("pthread_mutex_unlock()\n", rc);
   return NULL;
}


int main(int argc, char **argv)
```

```
{
   pthread_t              thread[NUMTHREADS];
   int                    rc=0;
   int                    amountOfData=4;
   int                    i;

   printf("Enter Testcase - %s\n", argv[0]);

   printf("Create/start threads\n");
   for (i=0; i <NUMTHREADS; ++i) {
rc = pthread_create(&thread[i], NULL, theThread, NULL);
      checkResults("pthread_create()\n", rc);
   }

   /* The producer loop */
   while (amountOfData--) {
      printf("Producer: 'Finding' data\n");
      sleep(3);

      rc = pthread_mutex_lock(&dataMutex);    /* Protect shared data and flag  */
      checkResults("pthread_mutex_lock()\n", rc);
      printf("Producer: Make data shared and notify consumer\n");
      ++sharedData;                           /* Add data                      */
      dataPresent=1;                          /* Set boolean predicate         */

      rc = pthread_cond_signal(&dataPresentCondition); /* wake up a consumer  */
      if (rc) {
         pthread_mutex_unlock(&dataMutex);
         printf("Producer: Failed to wake up consumer, rc=%d\n", rc);
         exit(1);
      }

      printf("Producer: Unlock shared data and flag\n");
      rc = pthread_mutex_unlock(&dataMutex);
      checkResults("pthread_mutex_lock()\n",rc);
   }

   printf("Wait for the threads to complete, and release their resources\n");
   for (i=0; i <NUMTHREADS; ++i) {
rc = pthread_join(thread[i], NULL);
      checkResults("pthread_join()\n", rc);
   }

   printf("Clean up\n");
   rc = pthread_mutex_destroy(&dataMutex);
   rc = pthread_cond_destroy(&dataPresentCondition);
   printf("Main completed\n");
   return 0;
}
```

Example: Using condition variables in Java programs:

This example shows a Java program using condition variables in the form of the wait and notify methods on a Java object. Note the locking protocol used.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
FileName: ATEST18.java
The output of this example is as follows:
 Entered the testcase
 Create/start the thread
 Consumer Thread-1: Entered
 Consumer Thread-1: Wait for the data to be produced
 Producer: 'Finding data
```

```
Consumer Thread-2: Entered
Consumer Thread-2: Wait for the data to be produced
Producer: Make data shared and notify consumer
Producer: Unlock shared data and flag
Consumer Thread-2: Found data or notified, CONSUME IT while holding inside the monitor
Consumer Thread-2: Wait for the data to be produced
Producer: 'Finding data
Producer: Make data shared and notify consumer
Producer: Unlock shared data and flag
Producer: 'Finding data
Consumer Thread-2: Found data or notified, CONSUME IT while holding inside the monitor
Consumer Thread-2: All done
Producer: Make data shared and notify consumer
Producer: Unlock shared data and flag
Producer: 'Finding data
Consumer Thread-1: Found data or notified, CONSUME IT while holding inside the monitor
Consumer Thread-1: Wait for the data to be produced
Producer: Make data shared and notify consumer
Producer: Unlock shared data and flag
Wait for the threads to complete
Consumer Thread-1: Found data or notified, CONSUME IT while holding inside the monitor
Consumer Thread-1: All done
Testcase completed
*/
import java.lang.*;

/* This class is an encapsulation of the condition variable plus */
/* mutex locking logic that can be seen in the Pthread example    */
class theSharedData extends Object {
    int                     dataPresent;
    int                     sharedData;

    public theSharedData() {
        dataPresent=0;
        sharedData=0;
    }
    public synchronized void get() {
        while (dataPresent == 0) {
            try {
                System.out.print("Consumer " +
                            Thread.currentThread().getName() +
                            ": Wait for the data to be produced\n");
                wait();
            }
            catch (InterruptedException e) {
                System.out.print("Consumer " +
                            Thread.currentThread().getName() +
                            ": wait interrupted\n");
            }
        }
        System.out.print("Consumer " +
                        Thread.currentThread().getName() +
                        ": Found data or notified, CONSUME IT " +
                        "while holding inside the monitor\n");
        --sharedData;
        if (sharedData == 0) {dataPresent=0;}
        /* in a real world application, the actual data would be returned */
        /* here                                                          */
    }
    public synchronized void put() {
        System.out.print("Producer: Make data shared and notify consumer\n");
        ++sharedData;
        dataPresent=1;
        notify();
        System.out.print("Producer: Unlock shared data and flag\n");
        /* unlock occurs when leaving the synchronized method */
    }
```

```
}

public class ATEST18 {
   public final static int NUMTHREADS    = 2;
   public static theSharedData dataConditionEncapsulation = new theSharedData();

   static class theThread extends Thread {
      public void run() {
         int        retries=2;

         System.out.print("Consumer " + getName() + ": Entered\n");
         while (retries-- != 0) {
            dataConditionEncapsulation.get();
            /* Typically an application would process the data outside */
            /* the monitor (synchronized get method here)              */
         }
         System.out.print("Consumer " + getName() + ": All done\n");
      }
   }

   public static void main(String argv[]) {
      int              amountOfData = 4;
      theThread threads[] = new theThread[NUMTHREADS];
      System.out.print("Entered the testcase\n");

      System.out.print("Create/start the thread\n");
      for (int i=0; i <NUMTHREADS; ++i) {
threads[i] = new theThread();
         threads[i].start();
      }

      while (amountOfData-- != 0) {
         System.out.print("Producer: 'Finding data\n");
         try {
            Thread.sleep(3000);
         }
         catch (InterruptedException e) {
            System.out.print("sleep interrupted\n");
         }
         dataConditionEncapsulation.put();
      }

      System.out.print("Wait for the threads to complete\n");
      for(int i=0; i <NUMTHREADS; ++i) {
  try {
            threads[i].join();
         }
         catch (InterruptedException e) {
            System.out.print("Join interrupted\n");
         }
      }

      System.out.print("Testcase completed\n");
      System.exit(0);
   }
}
```

Threads as synchronization primitives

Threads themselves can be used as synchronization primitives when one thread specifically waits for another thread to complete.

The waiting thread does not continue processing until the target thread has finished running all of its application code. Compared with other synchronization techniques, there is little cooperation in this synchronization mechanism.

A thread that is used as a synchronization primitive does not have the concept of an owner, such as in other synchronization techniques. A thread just waits for another to finish processing and end.

Related concepts

"Ending a thread" on page 17
When a thread has completed its processing, it takes an action to end itself and release system resources for use by other threads.

Space location locks

A space location lock puts a logical lock on any single byte of storage. The lock does not change the storage or effect your application's access to the storage. The lock is a piece of information recorded by the system.

Space location locks provide cooperative locking similar to that provided by mutual exclusion (mutex). However, space location locks differ from mutex in several respects:

- You can use the space location lock directly on the data it is protecting. The space location lock does not require your application to create and maintain an additional object. Correct results in your application still depend on the fact that all threads that are accessing the data use the space location lock.
- Space location locks allow an application to coordinate the use of different locking request types. For example, more than one thread can use space location locks to acquire a shared lock on the same data.
- Due to the extra lock types that are provided by space location locks, the concept of an owner for space location locks is slightly different from that for mutexes. There can be multiple owners of a shared lock if each owner has successfully acquired the shared lock. For a thread to get an exclusive lock, all of the shared locks must be unlocked.
- Space location locks have different performance implications to your application from those to mutexes. A space location lock requires about 500 reduced instruction set computer (RISC) instructions to lock in a path without contention between other threads. A mutex requires about 50 RISC instructions in the same path. However, a space location lock does not require any instructions for creation or deletion, whereas a mutex requires approximately 1000 RISC instructions for creation or deletion.

Note: Java does not have the ability to directly use space location locks. Space location locks require the use of pointers.

Example: Space location locks in Pthread programs:

This example shows a Pthread program that dynamically initializes an integer using a space location lock for synchronization.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
Filename: ATEST19.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST19
 Hold Lock to prevent access to shared data
 Create/start threads
 Thread 00000000 00000025: Entered
 Wait a bit until we are 'done' with the shared data
 Thread 00000000 00000026: Entered
 Thread 00000000 00000027: Entered
 Unlock shared data
 Wait for the threads to complete, and release their resources
 Thread 00000000 00000025: Start critical section, holding lock
 Thread 00000000 00000025: End critical section, release lock
 Thread 00000000 00000026: Start critical section, holding lock
 Thread 00000000 00000026: End critical section, release lock
```

```
 Thread 00000000 00000027: Start critical section, holding lock
 Thread 00000000 00000027: End critical section, release lock
 Main completed
*/
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <mih/milckcom.h>      /* Lock types            */
#include <mih/locksl.h>        /* LOCKSL instruction    */
#include <mih/unlocksl.h>      /* UNLOCKSL instruction */

#define checkResults(string, val) {                \
 if (val) {                                        \
   printf("Failed with %d at %s", val, string); \
   exit(1);                                        \
 }                                                 \
}

#define                NUMTHREADS    3
int                    sharedData=0;
int                    sharedData2=0;

void *theThread(void *parm)
{
   int    rc;
   printf("Thread %.8x %.8x: Entered\n", pthread_getthreadid_np());
   locksl(&sharedData, _LENR_LOCK);   /* Lock Exclusive, No Read */
   /********** Critical Section *******************/
   printf("Thread %.8x %.8x: Start critical section, holding lock\n",
          pthread_getthreadid_np());
   /* Access to shared data goes here */
   ++sharedData; --sharedData2;
   printf("Thread %.8x %.8x: End critical section, release lock\n",
          pthread_getthreadid_np());
   unlocksl(&sharedData, _LENR_LOCK); /* Unlock Exclusive, No Read */
   /********** Critical Section *******************/
   return NULL;
}

int main(int argc, char **argv)
{
  pthread_t             thread[NUMTHREADS];
  int                   rc=0;
  int                   i;

  printf("Enter Testcase - %s\n", argv[0]);

  printf("Hold Lock to prevent access to shared data\n");
  locksl(&sharedData, _LENR_LOCK);    /* Lock Exclusive, No Read */

  printf("Create/start threads\n");
  for (i=0; i <NUMTHREADS; ++i) {
     rc = pthread_create(&thread[i], NULL, theThread, NULL);
     checkResults("pthread_create()\n", rc);
  }

  printf("Wait a bit until we are 'done' with the shared data\n");
  sleep(3);
  printf("Unlock shared data\n");
  unlocksl(&sharedData, _LENR_LOCK); /* Unlock Exclusive, No Read */

  printf("Wait for the threads to complete, and release their resources\n");
  for (i=0; i <NUMTHREADS; ++i) {
 rc = pthread_join(thread[i], NULL);
     checkResults("pthread_join()\n", rc);
```

```
  }

  printf("Main completed\n");
  return 0;
}
```

Related concepts

"One-time initialization and thread safety"
At times, you might want to defer the initialization of resources until a thread requires them.
However, your application might require that multiple threads use a certain resource, which requires
you to initialize the resource only once in a threadsafe way.

Object locks

Object locks provide ways to acquire locks on specific systems or application objects.

In some cases, the system acquires object locks on behalf of actions that a user takes against certain
objects. The system enforces object locks for some actions.

You can acquire object locks such that the lock is effective only within the thread (thread-scoped), or
within the process (process-scoped). If two threads in the same process each try to acquire a
process-scoped lock to a system object, that lock is satisfied for both threads. Neither thread prevents the
other from acquiring the lock if they are in the same process.

If you are using object locks to protect access to an object from two threads within the same process, you
must use object locks that are scoped to a thread. A thread-scoped object lock never conflicts with an
object lock scoped to a process that is acquired by the same process.

Object locks allow an application to coordinate the use of different locking request types. More than one
thread can acquire a shared but thread-scoped lock on the same system object. Your application can also
acquire different types of object locks in a way that is similar to space locks. This allows more than one
thread to acquire a thread-scoped shared lock on a single target.

Because extra lock types are provided by object locks, the concept of an owner for object locks is slightly
different from that for mutual exclusions (mutexes). There can be multiple owners of a shared,
thread-scoped lock. If each owner has successfully acquired the shared thread-scoped lock, all of the
shared locks must be unlocked for a thread to get an exclusive thread-scoped object lock.

One-time initialization and thread safety

At times, you might want to defer the initialization of resources until a thread requires them. However,
your application might require that multiple threads use a certain resource, which requires you to
initialize the resource only once in a threadsafe way.

You can use several ways to initialize a resource that is used multiple times in a threadsafe fashion. Most
of these methods involve a Boolean value to allow the application to quickly determine whether the
required initialization is completed. You must also use a synchronization technique in addition to the
Boolean flag to ensure that the initialization is completed.

Related concepts

"Function calls that are not threadsafe" on page 54
Multithreaded application at times requires access to functions or system services that are not
threadsafe. There are few completely safe alternatives for calling these functions.

"Common multithreaded programming errors" on page 56
Several programming errors often occur when you write multithreaded applications.

Related reference

"Example: Space location locks in Pthread programs" on page 45
This example shows a Pthread program that dynamically initializes an integer using a space location
lock for synchronization.

"Data that is private to a thread" on page 49

You can use two methods to create data that is private to a thread: thread local storage and thread-specific data.

Example: One-time initialization in Pthread programs

This example shows a Pthread program that dynamically initializes an integer using the one-time initialization support.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
Filename: ATEST20.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST20
 Create/start threads
 Wait for the threads to complete, and release their resources
 Thread 00000000 00000007: Entered
 Thread 00000000 00000000: INITIALIZE RESOURCE
 Thread 00000000 00000007: The resource is 42
 Thread 00000000 00000006: Entered
 Thread 00000000 00000009: Entered
 Thread 00000000 00000008: Entered
 Thread 00000000 0000000a: Entered
 Thread 00000000 00000006: The resource is 42
 Thread 00000000 0000000a: The resource is 42
 Thread 00000000 00000009: The resource is 42
 Thread 00000000 00000008: The resource is 42
 Main completed
*/
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>t
#include <unistd.h>

#define checkResults(string, val) {              \
 if (val) {                                      \
   printf("Failed with %d at %s", val, string); \
   exit(1);                                      \
 }                                               \
}

#define              NUMTHREADS   5
pthread_once_t       oneTimeInit = PTHREAD_ONCE_INIT;
int                  initialized = 0;
int                  resource    = 0;

void initFunction(void)
{
   printf("Thread %.8x %.8x: INITIALIZE RESOURCE\n");
   resource = 42;
   /* Ensure that all initialization is complete and flushed */
   /* to storage before turning on this boolean flag         */
   /* Perhaps call a function or register an exception        */
   /* that causes an optimization boundary                    */
   initialized = 1;
}

void *theThread(void *parm)
{
   int   rc;
   printf("Thread %.8x %.8x: Entered\n", pthread_getthreadid_np());
   if (!initialized) {
      rc = pthread_once(&oneTimeInit, initFunction);
      checkResults("pthread_once()\n", rc);
```

```
  }
  printf("Thread %.8x %.8x: The resource is %d\n",
         pthread_getthreadid_np(), resource);
  return NULL;
}

int main(int argc, char **argv)
{
  pthread_t           thread[NUMTHREADS];
  int                 rc=0;
  int                 i;

  printf("Enter Testcase - %s\n", argv[0]);

  printf("Create/start threads\n");
  for (i=0; i <NUMTHREADS; ++i) {
  rc = pthread_create(&thread[i], NULL, theThread, NULL);
     checkResults("pthread_create()\n", rc);
  }

  printf("Wait for the threads to complete, and release their resources\n");
  for (i=0; i <NUMTHREADS; ++i) {
  rc = pthread_join(thread[i], NULL);
     checkResults("pthread_join()\n", rc);
  }

  printf("Main completed\n");
  return 0;
}
```

Data that is private to a thread

You can use two methods to create data that is private to a thread: thread local storage and thread-specific data.

Thread local storage

Thread local storage is similar to thread-specific storage, except that thread local storage is implemented by the operating system, whereas thread-specific storage is implemented with APIs. As a result, thread local storage is faster and easier to use than thread-specific storage.

ILE C, ILE C++, and ILE RPG compilers support thread local storage.

Thread-specific data

Thread-specific data allows a thread to maintain its own global storage that is hidden from the other threads.

When changing the application or application services to run in a multithreaded application, you must use a synchronization technique to protect global storage from being changed by multiple threads at the same time.

Due to the design of the application, threads might not function correctly if they share the global storage of the application. If eliminating the global storage is not feasible, you should consider using thread-specific data.

Consider the example of a server that stores information about the client and the current transaction in global storage. This server is never able to share the client information in a multithreaded environment without significant redesign. The application can instead pass the client information from function to function instead of using the global client information.

However, the application can be modified to use thread-specific data more easily than it can be modified to eliminate the use of global storage. When each new thread is created, the thread uses a global identifier (or key) to create and store its thread-specific data. Each client (thread) then has unique but global client data.

In addition, some APIs provide a way for the system to automatically call a *data destructor* function that cleans up the thread-specific data when a thread ends.

Related concepts

"One-time initialization and thread safety" on page 47
At times, you might want to defer the initialization of resources until a thread requires them. However, your application might require that multiple threads use a certain resource, which requires you to initialize the resource only once in a threadsafe way.

Related reference

WebSphere Development Studio: ILE C/C++ Language Reference

Websphere Development Studio: ILE RPG Reference

Example: Thread local storage in a Pthread program

This example shows a Pthread program using thread local storage.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
Filename: ATEST22TLS.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST22TLS
 Create/start threads
 Wait for the threads to complete, and release their resources
 Thread 0000000000000036: Entered
 Thread 0000000000000037: Entered
 Thread 0000000000000036: foo(), TLS data=0 2
 Thread 0000000000000036: bar(), TLS data=0 2
 Thread 0000000000000037: foo(), TLS data=1 4
 Thread 0000000000000037: bar(), TLS data=1 4
 Main completed
*/
#define _MULTI_THREADED
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void foo(void);  /* Functions that use the TLS data */
void bar(void);

#define checkResults(string, val) {             \
 if (val) {                                     \
   printf("Failed with %d at %s", val, string); \
   exit(1);                                     \
 }                                              \
}

__thread int TLS_data1;
__thread int TLS_data2;

#define                 NUMTHREADS    2

typedef struct {
   int    data1;
   int    data2;
} threadparm_t;
```

```
| void *theThread(void *parm)
| {
|     int             rc;
|     threadparm_t    *gData;
|
|     printf("Thread %.16llx: Entered\n", pthread_getthreadid_np());
|
|     gData = (threadparm_t *)parm;
|
|     TLS_data1 = gData->data1;
|     TLS_data2 = gData->data2;
|
|     foo();
|     return NULL;
| }
|
| void foo() {
|     printf("Thread %.16llx: foo(), TLS data=%d %d\n",
|            pthread_getthreadid_np(), TLS_data1, TLS_data2);
|     bar();
| }
|
| void bar() {
|     printf("Thread %.16llx: bar(), TLS data=%d %d\n",
|            pthread_getthreadid_np(), TLS_data1, TLS_data2);
|     return;
| }
|
|
| int main(int argc, char **argv)
| {
|     pthread_t          thread[NUMTHREADS];
|     int                rc=0;
|     int                i;
|     threadparm_t       gData[NUMTHREADS];
|
|     printf("Enter Testcase - %s\n", argv[0]);
|
|     printf("Create/start threads\n");
|     for (i=0; i < NUMTHREADS; i++) {
|         /* Create per-thread TLS data and pass it to the thread */
|         gData[i].data1 = i;
|         gData[i].data2 = (i+1)*2;
|         rc = pthread_create(&thread[i], NULL, theThread, &gData[i]);
|         checkResults("pthread_create()\n", rc);
|     }
|
|     printf("Wait for the threads to complete, and release their resources\n");
|     for (i=0; i < NUMTHREADS; i++) {
|         rc = pthread_join(thread[i], NULL);
|         checkResults("pthread_join()\n", rc);
|     }
|
|     printf("Main completed\n");
|     return 0;
| }
```

Example: Thread-specific data in a Pthread program

This example shows a Pthread program creating thread-specific data.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
Filename: ATEST22.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST22
```

```
 Create/start threads
 Wait for the threads to complete, and release their resources
 Thread 00000000 00000036: Entered
 Thread 00000000 00000036: foo(), threadSpecific data=0 2
 Thread 00000000 00000036: bar(), threadSpecific data=0 2
 Thread 00000000 00000036: Free data
 Thread 00000000 00000037: Entered
 Thread 00000000 00000037: foo(), threadSpecific data=1 4
 Thread 00000000 00000037: bar(), threadSpecific data=1 4
 Thread 00000000 00000037: Free data
 Main completed
*/
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void foo(void);  /* Functions that use the threadSpecific data */
void bar(void);
void dataDestructor(void *data);

#define checkResults(string, val) {              \
 if (val) {                                      \
   printf("Failed with %d at %s", val, string); \
   exit(1);                                      \
 }                                               \
}

typedef struct {
  int         threadSpecific1;
  int         threadSpecific2;
} threadSpecific_data_t;

#define                 NUMTHREADS    2
pthread_key_t           threadSpecificKey;


void *theThread(void *parm)
{
   int              rc;
   threadSpecific_data_t     *gData;
   printf("Thread %.8x %.8x: Entered\n", pthread_getthreadid_np());
   gData = (threadSpecific_data_t *)parm;
   rc = pthread_setspecific(threadSpecificKey, gData);
   checkResults("pthread_setspecific()\n", rc);
   foo();
   return NULL;
}

void foo() {
   threadSpecific_data_t *gData = pthread_getspecific(threadSpecificKey);
   printf("Thread %.8x %.8x: foo(), threadSpecific data=%d %d\n",
          pthread_getthreadid_np(), gData->threadSpecific1, gData->threadSpecific2);
   bar();
}

void bar() {
   threadSpecific_data_t *gData = pthread_getspecific(threadSpecificKey);
   printf("Thread %.8x %.8x: bar(), threadSpecific data=%d %d\n",
          pthread_getthreadid_np(), gData->threadSpecific1, gData->threadSpecific2);
   return;
}

void dataDestructor(void *data) {
   printf("Thread %.8x %.8x: Free data\n", pthread_getthreadid_np());
   pthread_setspecific(threadSpecificKey, NULL);
```

```
    free(data);
}


int main(int argc, char **argv)
{
  pthread_t              thread[NUMTHREADS];
  int                    rc=0;
  int                    i;
  threadSpecific_data_t       *gData;

  printf("Enter Testcase - %s\n", argv[0]);
  rc = pthread_key_create(&threadSpecificKey, dataDestructor);
  checkResults("pthread_key_create()\n", rc);

  printf("Create/start threads\n");
  for (i=0; i <NUMTHREADS; i++) {
      /* Create per-thread threadSpecific data and pass it to the thread */
    gData = (threadSpecific_data_t *)malloc(sizeof(threadSpecific_data_t));
    gData->threadSpecific1 = i;
    gData->threadSpecific2 = (i+1)*2;
    rc = pthread_create(&thread[i], NULL, theThread, gData);
    checkResults("pthread_create()\n", rc);
  }

  printf("Wait for the threads to complete, and release their resources\n");
  for (i=0; i <NUMTHREADS; i++) {
  rc = pthread_join(thread[i], NULL);
    checkResults("pthread_join()\n", rc);
  }

  pthread_key_delete(threadSpecificKey);
  printf("Main completed\n");
  return 0;
}
```

Example: Thread-specific data in a Java program

This example shows a Java program creating thread-specific data. Because a Java thread is created on an object, the use of thread-specific data is transparent.

Java is a language that performs garbage collection. Note the lack of data destructors or other cleanup action.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
FileName: ATEST22.java
The output of this example is as follows:
 Entered the testcase
 Create/start threads
 Thread Thread-1: Entered
 Thread Thread-1: foo(), threadSpecific data=0 2
 Thread Thread-1: bar(), threadSpecific data=0 2
 Wait for the threads to complete
 Thread Thread-2: Entered
 Thread Thread-2: foo(), threadSpecific data=1 4
 Thread Thread-2: bar(), threadSpecific data=1 4
 Testcase completed
*/
import java.lang.*;


public class ATEST22 {
   public final static int NUMTHREADS = 2;
```

```
    static class theThread extends Thread {
        int                threadSpecific1;
        int                threadSpecific2;

        public theThread(int i, int i2) {
            threadSpecific1 = i;
            threadSpecific2 = i2;
        }
        public void run() {
            System.out.print("Thread " + getName() +
                             ": Entered\n");
            foo();
            return;
        }
        void foo() {
            System.out.print("Thread " + getName() +
                             ": foo(), threadSpecific data=" +
                             String.valueOf(threadSpecific1) + " " +
                             String.valueOf(threadSpecific2) + "\n");
            bar();
        }
        void bar() {
            System.out.print("Thread " + getName() +
                             ": bar(), threadSpecific data=" +
                             String.valueOf(threadSpecific1) + " " +
                             String.valueOf(threadSpecific2) + "\n");
        }
    }

    public static void main(String argv[]) {
        System.out.print("Entered the testcase\n");

        System.out.print("Create/start threads\n");
        theThread threads[] = new theThread[NUMTHREADS];
        for (int i=0; i <NUMTHREADS; ++i) {
 threads[i] = new theThread(i, (i+1)*2);
            threads[i].start();
        }

        System.out.print("Wait for the threads to complete\n");
        for (int i=0; i <NUMTHREADS; ++i) {
  try {
            threads[i].join();
          }
          catch (InterruptedException e) {
              System.out.print("Join interrupted\n");
          }
        }
        System.out.print("Testcase completed\nj");
        System.exit(0);
    }

}
```

Function calls that are not threadsafe

Multithreaded application at times requires access to functions or system services that are not threadsafe. There are few completely safe alternatives for calling these functions.

To illustrate the alternatives, consider the example of a program that calls API foo(). Because the function foo() is listed as not threadsafe, you must find a safe way to call it. Two common options are using a global mutual exclusion (mutex) or using a separate job to run the function that is not threadsafe.

Related concepts

"Thread safety" on page 28
A function is threadsafe if you can start it simultaneously in multiple threads within the same process. A function is threadsafe only if all the functions it calls are also threadsafe.

"Synchronization techniques among threads" on page 33
When you create code that is threadsafe but still benefits from sharing data or resources between threads, the most important aspect of programming becomes the ability to synchronize threads.

"One-time initialization and thread safety" on page 47
At times, you might want to defer the initialization of resources until a thread requires them. However, your application might require that multiple threads use a certain resource, which requires you to initialize the resource only once in a threadsafe way.

"Common multithreaded programming errors" on page 56
Several programming errors often occur when you write multithreaded applications.

Related reference

"CL commands and thread safety" on page 30
The Integrated Language Environment (ILE) CL run time and compiler-generated code are threadsafe. Original program model (OPM) CL programs are not threadsafe.

"Threads considerations for ILE COBOL language" on page 60
An Integrated Language Environment (ILE) COBOL module that runs safely in a multithreaded environment can be created by serializing access to the module.

Using global mutexes to run functions that are not threadsafe

To call a function that is not threadsafe, you can use a global mutual exclusion (mutex).

You might try locking a mutex that you call the FOO MUTEX whenever you call foo(). This works if you know everything about the internal parts of foo(), but for general operating system functions, this is still not a completely safe solution.

For example, API foo() is not threadsafe because it calls API bar(), which calls threadUnsafeFoo(). The threadUnsafeFoo() function uses certain storage and control blocks in an unsafe way. In this example, those storage and control blocks are called DATA1.

Because the API threadUnsafeFoo() is unsafe, any application or system service that uses foo() is not threadsafe because using it with multiple threads might result in damaged data or undefined results.

Your application also uses a threadsafe API or system service that is called wiffle(). Wiffle() calls waffle(), which accesses storage and control blocks DATA1 in a threadsafe manner. (That is waffle() and a group of other APIs use an internal DATA1 MUTEX).

Your application does not know the underlying DATA1 connection between foo() and wiffle() system services. Your application, therefore, does not lock the FOO MUTEX before calling wiffle(). When a call to foo() (holding the FOO MUTEX) occurs at the same time as a call to wiffle(), the application's use of the function foo() that is not threadsafe causes DATA1 and whatever it represents to be damaged.

As complex as this scenario is, consider how complex resolving failures is when you do not know the internal details of all the functions involved.

Do not try to make operating system services or functions threadsafe by using your own serialization.

Using separate jobs to run functions that are not threadsafe

There are several threadsafe mechanisms to submit a new job to complete the processing you need.

- If your application uses CL commands that are not threadsafe, you can use the system API Qp0zSystem(). This API is similar to the C system() function; however, it starts a new process before running the CL command. It also returns some information about the results of the CL command that you ran. For more details about Qp0zSystem(), see the System API Programming Reference.

- If your application calls APIs or programs that are not threadsafe, you might use the spawn() API to start a new job. If you use the spawn() API, your application must decide which part of the application environment gets duplicated into the child. The spawn() API allows the child process to inherit open IFS files, sockets, or other resources from the parent. For details about spawn(), see the System API Programming Reference.
- Your application might frequently use multiple functions that are not threadsafe. In this case, you should use one of the previously mentioned mechanisms to start a new job. That new job can function as a server, running the code that is not threadsafe when requested by your application. Your application can use message queues, data queues, or perhaps semaphores to communicate with your server.

When using any of these separate job mechanisms, be aware of the following problems involved with using a different job to complete some of your application processing:

- Input/output considerations

  The server job cannot perform normal I/O processing without conflicting with the application I/O processing. The application I/O processing can affect file offsets or locks that are maintained on system objects. Attempts at I/O processing can conflict between the jobs.

- Parameter passing

  Passing pointers or other types of complex parameters to a function that is not threadsafe might be difficult to accomplish. That function runs in a separate job. You need to solve the problems involved with passing complex data to that function.

- Returning function calls

  The results from the function that is not threadsafe might be more complex than the simple pass or fail type of information that is provided by some of the mechanisms described previously. This case is similar to the parameter passing problem described previously.

- Job attributes

  You might have additional complexity to manage when you try to duplicate your application's environment in the server job. For example, if you started your application as user ALICE, and you started the server job as user BOB, the application behavior is not correct.

Common multithreaded programming errors

Several programming errors often occur when you write multithreaded applications.

Related concepts

"One-time initialization and thread safety" on page 47
At times, you might want to defer the initialization of resources until a thread requires them. However, your application might require that multiple threads use a certain resource, which requires you to initialize the resource only once in a threadsafe way.

"Function calls that are not threadsafe" on page 54
Multithreaded application at times requires access to functions or system services that are not threadsafe. There are few completely safe alternatives for calling these functions.

"Debugging and servicing of multithreaded jobs" on page 61
These concepts and techniques can help you with debugging and servicing multithreaded jobs.

Related information

Troubleshooting pthread errors

Calling functions that are not threadsafe

When you write a threaded application, the most common programming error is the use of APIs or system services that are not threadsafe.

An application must be aware of each API that it calls and whether the provider classifies the API as threadsafe. An API can be classified as threadsafe if the API only calls other threadsafe APIs or system services.

This is especially problematic when your application calls user-written code that is outside of your control. You cannot validate this code for thread safety. If the user code is not threadsafe and you call it within your application's own process context, you might damage the application data, or the application might stop.

Related reference

"Threads considerations for ILE C and ILE C++ languages" on page 59
You must evaluate the existing Integrated Language Environment (ILE) C and ILE C ++ applications for thread safety before calling them in a multithreaded job. If your program is not threadsafe, techniques are available for calling programs that are not threadsafe from multithreaded jobs.

Thread creation not allowed failure

Not all jobs in the i5/OS operating system are allowed to create threads.

When you create a job, you can specify special parameters that indicate that the job allows multiple threads. For example, you can specify the multithreaded flag in the inheritance parameter when the spawn() application programming interface (API) is used to create a job.

For some job types, you can use an attribute of the job description to specify that jobs started by using the job description allow threads. To do this, use the Allow Multiple Threads (ALWMLTTHD) parameter on the Create Job Description (CRTJOBD) or on the Change Job Description (CHGJOBD) command. You can also specify the ALWMLTTHD parameter on the Submit Job (SBMJOB) command to allow threads to be created even if the job description being used on the SBMJOB command does not have ALWMLTTHD(*YES) specified.

If you attempt to create a thread in a job that is not capable of starting threads, thread creation fails. See your API or language specific documentation for details on how the failure is manifested.

You will also see a thread creation not allowed failure if your application is running the destructors for C++ static objects or C/C++ program termination routines. Application code cannot create threads when the job is ending.

Ending of activation group

The i5/OS Integrated Language Environment (ILE) program model uses activation groups as a way to encapsulate resources for an application program within a job.

When multiple threads exist in the processes that possibly use the activation group, the system cannot end that activation group in a predictable and safe way. To solve this problem, the system changes the behavior of ending an activation group in a multithreaded job.

If a secondary thread in a multithreaded application performs an action that ends an activation group (for example, C functions exit() or abort()), or fails to prevent an activation group from ending, the system takes action to end the process at that time. If the initial thread in a multithreaded application performs an action that ends an activation group, or fails to prevent an activation group from ending, the system does not end the process.

A typical example of this occurs when a current i5/OS application is altered to take advantage of multithreading. The current i5/OS application probably uses multiple program objects. Unlike other platforms, i5/OS allows you to call the programs within the same process context as the currently running application code.

The default activation group attribute for a program is *NEW. Calling a program with a default activation of *NEW in a multithreaded job ends both the activation group and the process when the program ends. If the job is to remain active after the program ends, you must change the program to use a named activation group.

Mixing thread models

Do not mix Pthreads APIs with other thread management APIs that might be provided by the system. It might cause unpredictable results.

For example, you should not use Java or the IBM Open Class® libraries threads implementations in the same thread or process as Pthreads. More importantly, do not attempt to manipulate a thread from one threads implementation with the API of another. Unpredictable results might occur.

For example, you can manipulate the priority of a thread by using Pthread interfaces and non-Pthread interfaces. If you do so, the priority is always set correctly. However, the priority returned from the Pthread interface pthread_getschedparam() is correct only if setting the priority was always done using either the pthread_setschedparam() interface or some other interface, but not both. If you instead use multiple interfaces to set the priority of the thread, pthread_getschedparam() always returns the priority that was set by the last use of the pthread_setschedparam() interface.

Similarly, you can end a Java thread using the pthread_exit() API if you are running application code in a native method. However, using pthread_exit() to end the Java thread can have unexpected results. These results include bypassing some of the Java environment thread processing, or perhaps ending the Java thread in a way that your Java application does not expect and cannot handle.

Commit operations in multithreaded programs

Database transactions in the i5/OS operating system are scoped to the job or the activation group.

If your multithreaded application is working on multiple database transactions in multiple threads simultaneously, it is possible that a commit operation in one thread also commits activity done by another thread.

For example, an application has a thread working on a database transaction for client1 and another thread working on a database transaction for client2. The thread that is processing for client1 completes its processing and commits the changes. This commit operation also commits all the changes made for client2. Your application must be aware of the transactions that are in progress.

Related reference

"Database considerations for multithreaded programming" on page 6
When you use databases in multithreaded programs, be aware of these considerations.

Database record I/O and thread safety

The I/O feedback area of a database file can be used to communicate results of I/O operations to the record I/O user. A typical application might have problems with thread safety because of the nature of the feedback area.

When you perform an I/O operation, the support of the run time and of the database ensures that the I/O operation is threadsafe. When the I/O operation is completed, any locks that the run time holds are released and control is given back to the application.

It is the application's responsibility to serialize access when examining the I/O feedback area so that another thread does not use it simultaneously. COBOL and RPG use the feedback area on every I/O operation. Thus, if you perform I/O operations to the same shared file with different languages, you must serialize access to the file even if you are not apparently using the feedback area.

For example, Thread 1 uses file A for record I/O operations. The system stores information about those operations in the feedback area for file A. After the I/O operations for Thread 1 are completed, Thread 1 manipulates the feedback area by examining fields in the feedback area. If you do not protect the feedback area by serializing access to file A, Thread 2 might also use the same feedback area simultaneously when using file A for record I/O operations. As a result, Thread 1 can get inconsistent

results because Thread 2 is using the feedback area at the same time. This situation occurs when sharing a file is conducted by using the file pointer or file descriptor number. It is not a problem if the file is opened in each thread.

You must use a synchronization technique to protect the feedback area from being changed by other application code or by the system when other application code or the system accesses the feedback area.

> **Related reference**
>
> "Database considerations for multithreaded programming" on page 6
> When you use databases in multithreaded programs, be aware of these considerations.

Language access and threads

Knowing how different languages support threads on the i5/OS operating system can help you evaluate how threads can be implemented in your own application.

The concepts that are described in this section pertain to all programming languages. To determine how each language enables these concepts, refer to the Language topic in the information center for the specific language.

Threads considerations for Java language

Java threads operate on top of the i5/OS kernel threads model using the java.lang.Thread class. Each Java thread is one of the many tasks that run in the process.

You can do all of the activities that are listed in the Threads Management section.

The Java virtual machine (JVM) always creates several threads to perform services such as Java garbage collection. The system uses these threads; applications should not use them.

You can use native methods to access system functions that are not available in Java. Native methods are not *PGM objects. They are procedures that are exported from Integrated Language Environment (ILE) service programs (*SRVPGM). These native methods always run in multithreaded processes; therefore, they must be threadsafe. The ILE COBOL, RPG IV, CL, C, and C++ compilers are threadsafe.

Note: Not all standard C and C++ functions are threadsafe. Refer to the C and C++ topic in the information center.

When it is necessary to call an ILE program (*PGM) object, use java.lang.Runtime.exec() to start another process in which the program can run.

Use exit() and abort() with care. These functions end the application, including the process and all the threads that run in the process.

> **Related concepts**
>
> "Examples: Threads" on page 70
> These examples are used throughout the Multithreaded application topic collection.

Threads considerations for ILE C and ILE C++ languages

You must evaluate the existing Integrated Language Environment (ILE) C and ILE C ++ applications for thread safety before calling them in a multithreaded job. If your program is not threadsafe, techniques are available for calling programs that are not threadsafe from multithreaded jobs.

Not all ILE C/ C++ library functions are threadsafe. Before calling your application in a multithread-capable job, see *WebSphere Development Studio: ILE C/C++ Compiler Reference* PDF and *Standard C++ Library Reference* PDF to determine whether all of your functions are threadsafe.

When using programs that are written in ILE C or ILE C++, perform the following actions if necessary:

- Re-create ILE C or ILE C++ applications
  Compile and bind all existing ILE C or ILE C++ applications with TGTRLS(*CURRENT) before you call them in a multithreaded job.
- Eliminate *NEW activation groups
  Do not use *NEW activation groups in multithreaded applications. If you need to call an ILE C/C++ *PGM object that has an activation group of *NEW, either start another process to run the second program, or create the second program with a named activation group.

  Note: ACTGRP(*ENTMOD) is the default on the Create Program (CRTPGM) command. For ILE C and ILE C++, ACTGRP(*ENTMOD) causes a program to be created as if ACTGRP(*NEW) were specified. Programs created by using the Create Bound C Program (CRTBNDC) command or the Create Bound C++ Program (CRTBNDCPP) command have an activation group of *NEW.

  Related reference

  "Calling functions that are not threadsafe" on page 56
  When you write a threaded application, the most common programming error is the use of APIs or system services that are not threadsafe.

Threads considerations for ILE COBOL language

An Integrated Language Environment (ILE) COBOL module that runs safely in a multithreaded environment can be created by serializing access to the module.

You can serialize access to the ILE COBOL module by specifying THREAD (*SERIALIZE) on the PROCESS statement. After you serialize the module, only one thread can run any procedure in that module at one time. For example, consider a module that has procedures P1 and P2. If one thread is running procedure P1, no other thread can run either procedure P1 or P2 until the first thread finishes running P1.

Even when a module is serialized, the COBOL programmer must ensure that global storage and heap storage are accessed in a threadsafe way. Even if a COBOL procedure apparently uses only automatic storage, COBOL uses static storage control blocks in every procedure. Therefore, you must always specify THREAD (*SERIALIZE) when using ILE COBOL in a multithreaded environment.

  Related concepts

  "Function calls that are not threadsafe" on page 54
  Multithreaded application at times requires access to functions or system services that are not threadsafe. There are few completely safe alternatives for calling these functions.

Threads considerations for ILE RPG language

An Integrated Language Environment (ILE) RPG module that runs safely in a multithreaded environment can be created by creating the module to have thread-scoped static storage, or by serializing access to the module.

You can create the module to have all its static storage default to thread-scoped static storage by specifying THREAD(*CONCURRENT) on the Control specification.

You can serialize access to the module by specifying THREAD (*SERIALIZE) on the Control specification. After you serialize a module, only one thread can run any procedure in that module at one time. For

example, consider a module that has procedures P1 and P2. If one thread is running procedure P1, no other thread can run either procedure P1 or P2 until the first thread finishes running P1.

Even when a module uses thread-scoped static storage or the module is serialized, the RPG programmer must ensure that non-thread-scoped static storage, storage that is shared through import or export, and heap storage are accessed in a threadsafe way. Even if an RPG procedure apparently uses only automatic storage, RPG uses static storage control blocks in every procedure. Therefore, you must always specify THREAD(*CONCURRENT) or THREAD(*SERIALIZE) when using ILE RPG in a multithreaded environment.

Threads considerations for OPM language

Original program model (OPM) programs are not threadsafe. You should migrate OPM programs to Integrated Language Environment (ILE) and make them threadsafe before a multithreaded application calls them.

User-written Java methods should not call OPM programs. When it is necessary to call an OPM program in a multithreaded application, you should start another process to run the OPM program.

Related concepts

"Thread program models" on page 2
There are different models for creating multithread programs.

Debugging and servicing of multithreaded jobs

These concepts and techniques can help you with debugging and servicing multithreaded jobs.

The concepts that are described in this section pertain to all programming languages. To determine how each language enables these concepts, refer to the Language topic in the information center for the specific language.

Related concepts

"Common multithreaded programming errors" on page 56
Several programming errors often occur when you write multithreaded applications.

Commands that report thread-related data

Most of the commands used to service jobs are not changed to support multithreaded processes. Therefore, the existing commands continue to operate against service jobs, rather than individual threads within service jobs.

The following commands are enhanced to report thread-related data:

Dump Job (DMPJOB) command

> The Dump Job command now dumps all threads within a job. The thread-related data is available though use of the JOBTHD parameter on the command. The following example shows how to obtain a multithreaded job dump that contains the thread-related data:

```
STRSRVJOB JOB(000000/USER/JOBNAME)
DMPJOB PGM(*NONE) JOBARA(*NONE) ADROBJ(*NO) JOBTHD(*THDSTK)
ENDSRVJOB
```

Start Trace (STRTRC) command

> You can use the Start Trace command to trace either all threads in a multithreaded job, or up to 20 specific threads in a multithreaded job. The thread identifier is included in each trace record to indicate the thread that caused the record to be entered into the log. The following example shows how to obtain a trace for a multithreaded job:

```
STRTRC SSNID(MYSSNID) JOB((000000/USERNAME/JOBNAME (*ALL)))
    ... tracing the job
ENDTRC SSNID(MYSSNID) DTALIB(MYLIB) PRTTRC(*YES)
```

The following example shows how to obtain a trace for specified threads in a multithreaded job:

```
STRTRC SSNID(MYSSNID) JOB((000000/USERNAME/JOBNAME (00000001 00000002 00000003)))
    ... tracing the job
ENDTRC SSNID(MYSSNID) DTALIB(MYLIB) PRTTRC(*YES)
```

First Failure Data Capture (FFDC) function

FFDC is a function that you can use in your program, service program, or module to report a problem for an authorized program analysis report (APAR). FFDC logs the problem, builds a symptom string, and collects problem analysis data.

FFDC is not fully functional when used in a multithreaded program. When called in a multithreaded program, FFDC provides partial support that includes:

- Collecting data item information and storing this information in the QPSRVDMP spooled file.
- Storing the point of failure, symptom string, and the detecting and suspected program information in the QPSRVDMP spooled file. The job log also contains this information.
- Starting the Dump Job (DMPJOB) command.

FFDC support that is not provided within a multithreaded job includes object dumping, problem entry creation, and data collection for problem determination.

Flight recorders

A useful way to debug multithreaded applications is to write data to a flight recorder. A flight recorder is a file, output buffer, or some other object where trace information is written so that problems in the application can be tracked.

Entries to and exits from a function are typical information that is traced in a flight recorder. Parameters that are passed to functions, major changes in control flow, and error conditions are often tracked as well.

The Pthread library provides a way for your application to trace problems. You can use tracing to cut optional trace points, turn the recorder on or off, and recompile your application so that all of the tracing code is removed.

You can use different levels of tracing. If your application wants to honor the trace levels, it is the application's responsibility to cue trace points at the appropriate trace levels, or manage the trace level. The Pthread library provides APIs, macros, and trace level variables to assist you.

For Pthread tracing, an error-level tracing displays only those trace points that have error-level severity. An informational-level tracing displays the informational trace points in addition to all error-level trace points. A verbose-level tracing displays trace points of all levels. Each trace point automatically includes the thread ID, a timestamp of millisecond granularity, and the trace data.

The CL commands that enable, display, and work with the tracing are part of the i5/OS operating system.

The tracing buffers for your tracepoints are created as user space objects in the QUSRSYS library. The trace buffers are named QP0Zxxxxxx, in which xxxxxx is the six-digit job number of the job that performed the tracing.

The following APIs relate to tracing and are detailed in the UNIX-Type APIs guide:

- Qp0zUprintf - print formatted trace data
- Qp0zDump - dump formatted hexadecimal data
- Qp0zDumpStack - dump the call stack of the calling thread

- Qp0zDumpTargetStack - dump the call stack of the target thread
- Qp0zLprintf - print formatted job log message

The following CL commands that allow you to manipulate tracing:
- DMPUSRTRC - dump the contents of a specified job's trace
- CHGUSRTRC - change attributes (size, wrapping, clear) of a specified job's trace
- DLTUSRTRC - delete the persistent trace object that is associated with a job's trace

Example: Sample flight recorder output from a Pthread program

This example shows a Pthread program that uses the flight recorder or tracing interfaces that are provided by the system.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

```
/*
Filename: ATEST23.QCSRC
Use CL command DMPUSRTRC to output the following tracing
information that this example traces.

This information is put into a file QTEMP/QAP0ZDMP or to
standard output.

The trace records are indented and labeled based on thread id,
and millisecond timestamp of the time the tracepoint was cut.

The following trace output occurs when the optional parameter
'PTHREAD_TRACING' is NOT specified when calling this program.

If the optional parameter 'PTHREAD_TRACING' is specified, many
more tracepoints describing pthread library processing will occur.

Use the Pthread library tracepoints to debug incorrect calls to the
Pthreads library from your application.

Trace output ---------
User Trace Dump for job 096932/MYLIB/PTHREADT. Size: 300K, Wrapped 0 times.

--- 11/06/1998 11:06:57 ---
0000000D:133520 Create/start a thread
0000000D:293104 Wait for the thread to complete, and release their resources
0000000E:294072 Thread Entered
0000000E:294272 DB51A4C80A:001CD0 L:0008 Global Data
0000000E:294416  DB51A4C80A:001CD0  00000000 00000002                      *................*
0000000E:294496 foo(), threadSpecific data=0 2
0000000E:294568 bar(), threadSpecific data=0 2
0000000E:294624 bar(): This is an error tracepoint
0000000E:294680 Stack Dump For Current Thread
0000000E:294736 Stack:  This thread's stack at time of error in bar()
0000000E:333872 Stack:  Library    / Program     Module     Stmt     Procedure
0000000E:367488 Stack:  QSYS       / QLESPI       QLECRTTH    774    : LE_Create_Thread2__FP12crtth_parm_t
0000000E:371704 Stack:  QSYS       / QP0WPTHR     QP0WPTHR   1008    : pthread_create_part2
0000000E:371872 Stack:  MYLIB      / PTHREADT     PTHREADT     19    : theThread__FPv
0000000E:371944 Stack:  MYLIB      / PTHREADT     PTHREADT     29    : foo__Fv
0000000E:372016 Stack:  MYLIB      / PTHREADT     PTHREADT     46    : bar__Fv
0000000E:372104 Stack:  QSYS       / QP0ZCPA      QP0ZUDBG     87    : Qp0zDumpStack
0000000E:379248 Stack:  QSYS       / QP0ZSCPA     QP0ZSCPA    276    : Qp0zSUDumpStack
0000000E:379400 Stack:  QSYS       / QP0ZSCPA     QP0ZSCPA    287    : Qp0zSUDumpTargetStack
0000000E:379440 Stack:  Completed
0000000E:379560 foo(): This is an error tracepoint
0000000E:379656 dataDestructor: Free data
0000000D:413816 Create/start a thread
0000000D:414408 Wait for the thread to complete, and release their resources
0000000F:415672 Thread Entered
```

```
   0000000F:415872 DB51A4C80A:001CD0 L:0008 Global Data
   0000000F:416024  DB51A4C80A:001CD0  00000001 00000004                              *...............*
   0000000F:416104 foo(), threadSpecific data=1 4
   0000000F:416176 bar(), threadSpecific data=1 4
   0000000F:416232 bar(): This is an error tracepoint
   0000000F:416288 Stack Dump For Current Thread
   0000000F:416344 Stack:  This thread's stack at time of error in bar()
   0000000F:416552 Stack:  Library    / Program     Module    Stmt    Procedure
   0000000F:416696 Stack:  QSYS       / QLESPI       QLECRTTH   774    : LE_Create_Thread2__FP12crtth_parm_t
   0000000F:416784 Stack:  QSYS       / QP0WPTHR     QP0WPTHR  1008    : pthread_create_part2
   0000000F:416872 Stack:  MYLIB      / PTHREADT     PTHREADT    19    : theThread__FPv
   0000000F:416952 Stack:  MYLIB      / PTHREADT     PTHREADT    29    : foo__Fv
   0000000F:531432 Stack:  MYLIB      / PTHREADT     PTHREADT    46    : bar__Fv
   0000000F:531544 Stack:  QSYS       / QP0ZCPA      QP0ZUDBG    87    : Qp0zDumpStack
   0000000F:531632 Stack:  QSYS       / QP0ZSCPA     QP0ZSCPA   276    : Qp0zSUDumpStack
   0000000F:531704 Stack:  QSYS       / QP0ZSCPA     QP0ZSCPA   287    : Qp0zSUDumpTargetStack
   0000000F:531744 Stack:  Completed
   0000000F:531856 foo(): This is an error tracepoint
   0000000F:531952 dataDestructor: Free data
   0000000D:532528 Main completed
*/
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <qp0ztrc.h>
#define checkResults(string, val) {                \
 if (val) {                                        \
   printf("Failed with %d at %s", val, string); \
   exit(1);                                        \
 }                                                 \
}

typedef struct {
   int          threadSpecific1;
   int          threadSpecific2;
} threadSpecific_data_t;

#define                   NUMTHREADS    2
pthread_key_t             threadSpecificKey;

void foo(void);
void bar(void);
void dataDestructor(void *);

void *theThread(void *parm) {
   int                   rc;
   threadSpecific_data_t  *gData;
   PTHREAD_TRACE_NP({
                   Qp0zUprintf("Thread Entered\n");
                   Qp0zDump("Global Data", parm, sizeof(threadSpecific_data_t));},
                   PTHREAD_TRACE_INFO_NP);
   gData = (threadSpecific_data_t *)parm;
   rc = pthread_setspecific(threadSpecificKey, gData);
   checkResults("pthread_setspecific()\n", rc);
   foo();
   return NULL;
}

void foo() {
   threadSpecific_data_t *gData =
      (threadSpecific_data_t *)pthread_getspecific(threadSpecificKey);
   PTHREAD_TRACE_NP(Qp0zUprintf("foo(), threadSpecific data=%d %d\n",
                             gData->threadSpecific1, gData->threadSpecific2);,
                PTHREAD_TRACE_INFO_NP);
   bar();
```

```
      PTHREAD_TRACE_NP(Qp0zUprintf("foo(): This is an error tracepoint\n");,
                       PTHREAD_TRACE_ERROR_NP);
}

void bar() {
   threadSpecific_data_t *gData =
       (threadSpecific_data_t *)pthread_getspecific(threadSpecificKey);
   PTHREAD_TRACE_NP(Qp0zUprintf("bar(), threadSpecific data=%d %d\n",
                               gData->threadSpecific1, gData->threadSpecific2);,
                    PTHREAD_TRACE_INFO_NP);
   PTHREAD_TRACE_NP(Qp0zUprintf("bar(): This is an error tracepoint\n");
                    Qp0zDumpStack("This thread's stack at time of error in bar()");,
                    PTHREAD_TRACE_ERROR_NP);
   return;
}

void dataDestructor(void *data) {
   PTHREAD_TRACE_NP(Qp0zUprintf("dataDestructor: Free data\n");,
                    PTHREAD_TRACE_INFO_NP);
   pthread_setspecific(threadSpecificKey, NULL);    free(data);
   /* If doing verbose tracing we will even write a message to the job log */
   PTHREAD_TRACE_NP(Qp0zLprintf("Free'd the thread specific data\n");,
                    PTHREAD_TRACE_VERBOSE_NP);
}

/* Call this testcase with an optional parameter 'PTHREAD_TRACING' */
/* If the PTHREAD_TRACING parameter is specified, then the          */
/* Pthread tracing environment variable will be set, and the        */
/* pthread tracing will be re initialized from its previous value. */
/* NOTE: We set the trace level to informational, tracepoints cut   */
/*       using PTHREAD_TRACE_NP at a VERBOSE level will NOT show up*/
int main(int argc, char **argv) {
   pthread_t              thread[NUMTHREADS];
   int                    rc=0;
   int                    i;
   threadSpecific_data_t *gData;
   char                   buffer[50];

   PTHREAD_TRACE_NP(Qp0zUprintf("Enter Testcase - %s\n", argv[0]);,
                    PTHREAD_TRACE_INFO_NP);
   if (argc == 2 && !strcmp("PTHREAD_TRACING", argv[1])) {
      /* Turn on internal pthread function tracing support          */
      /* Or, use ADDENVVAR, CHGENVVAR CL commands to set this envvar*/
      sprintf(buffer, "QIBM_PTHREAD_TRACE_LEVEL=%d", PTHREAD_TRACE_INFO_NP);
      putenv(buffer);
      /* Refresh the Pthreads internal tracing with the environment */
      /* variables value.                                           */
      pthread_trace_init_np();
   }
   else {
      /* Trace only our application, not the Pthread code           */
      Qp0wTraceLevel = PTHREAD_TRACE_INFO_NP;
   }

   rc = pthread_key_create(&threadSpecificKey, dataDestructor);
   checkResults("pthread_key_create()\n", rc);

   for (i=0; i <NUMTHREADS; ++i) {
      /* Create per-thread threadSpecific data and pass it to the thread */
      gData = (threadSpecific_data_t *)malloc(sizeof(threadSpecific_data_t));
      gData->threadSpecific1 = i;
      gData->threadSpecific2 = (i+1)*2;
      rc = pthread_create( &thread[i], NULL, theThread, gData);
      checkResults("pthread_create()\n", rc);
      PTHREAD_TRACE_NP(Qp0zUprintf("Wait for the thread to complete, "
                                   "and release their resources\n");,
                       PTHREAD_TRACE_INFO_NP);
```

```
      rc = pthread_join(thread[i], NULL);
      checkResults("pthread_join()\n", rc);
   }

   pthread_key_delete(threadSpecificKey);
   PTHREAD_TRACE_NP(Qp0zUprintf("Main completed\n");,
                   PTHREAD_TRACE_INFO_NP);
   return 0;
}
```

Options to view thread information

You can use the Display Job (DSPJOB), Work with Job (WRKJOB), and Work with Active Jobs (WRKACTJOB) commands to display and work with threads associated with an i5/OS job.

Through each of these commands, you can select an option that you can use to display or work with the threads in the job. You can use the Work with Threads display to view the following information for the job:

- List of threads

   This display shows all of the threads that are currently associated with the job. The first thread in the list is the initial thread in the process. This list changes depending on the thread activity in the job. The 8-digit number is the thread identifier. Other information about this display includes the thread's status of active or waiting, total CPU usage, total auxiliary storage I/O counts, and thread run priority.

- Thread call stack option

   This option displays the call stack of any thread. In some conditions, typically when the thread is running an instruction that cannot be interrupted, the call stack cannot be displayed. In these cases, a message indicating that no stack information is available is displayed.

- Thread mutual exclusions (mutexes) option

   You can use this option to see all mutexes associated with a specific thread. This includes mutexes that are held and the mutex for which the thread might be waiting. If the specified thread has no mutexes associated with it, no mutex is displayed for the thread.

- Thread locks option

   You can use this option to see all the thread-scoped object locks held by the thread. You can also use this option to see all pending process-scoped locks and thread-scoped locks on which the thread is waiting. If the specified thread has no object locks associated with it, there is no lock displayed for the thread.

- Hold thread option

   You can use this option to temporarily suspend the running of a specific thread. This option is useful if you suspect that a thread might be looping or causing some other system problems. You must use the Hold thread option in conjunction with the Release thread option. Unlike Hold Job (HLDJOB), repeated Hold thread requests are cumulative. To resume the running of a held thread, you must issue the equivalent number of release thread requests.

- Release thread option

   You can use this option resume the running of a thread that was suspended.

- End thread option

   You can use this option to end a specific thread. In general, you must not use this option because it can cause unpredictable results. This option requires *SERVICE special authority.

The Release Job (RLSJOB), Hold Job (HLDJOB), End Job (ENDJOB), and End Job Abnormal (ENDJOBABN) commands affect all threads within a multithreaded job. No commands are planned to provide the equivalent thread-level support.

Multithreaded job debugging

You can use the mechanism provided by the spawn() API to start a debugging session before actually calling the multithreaded program to be debugged.

Setting the QIBM_CHILD_JOB_SNDINQMSG environment variable to 1 controls this capability.

An example SPAWN command is available for your use and modification as part of i5/OS option 7, the i5/OS Example Tools Library, QUSRTOOL. Member TP0ZINFO in the file QATTINFO in the QUSRTOOL library contains information about how to create the SPAWN CL command.

You can also simplify the process of starting a job that runs the multithreaded program and the debugging session by creating a command (such as the SPAWN command) that performs the necessary steps. The steps are as follows:

1. Sign on to a display station session
2. Start a job to run the multithreaded program
3. Use the Start Service Job (STRSRVJOB) command to service the job running the multithreaded program
4. Call the Start Debug (STRDBG) command, adding programs to debug, and setting breakpoints
5. Notify the servicing job when the thread hits a breakpoint so that it stops the thread
6. Stop all threads with the debugger support
7. Issue valid debug commands from the debug screen that is displayed
8. Resume the thread that hit the breakpoint and then all other threads
9. Repeat the above cycle until debugging is complete
10. Call the End Debug (ENDDBG) command

   Related reference

   API

   ILE Concepts PDF

   spawn() API

   SPAWN CL command, QUSRTOOL example

   WebSphere Development Studio: ILE C/C++ Programmer's Guide PDF

Areas to test for multithreaded applications

Testing is key to the process of verifying the correctness of a multithreaded program.

You must consider the following key concepts when testing a multithreaded program:

- Multiple instances

  Test your multithreaded program by having multiple instances of the program active at the same time. If your application allows the number of threads to vary, configure each instance of the program with a different number of threads.

- Varying system workload

  Test your multithreaded program by running it repeatedly with a different mix of applications running. The interaction of different applications might reveal timing problems or race conditions.

- Stressful environments

  Environments running stressful workload can reveal contention, timing, and performance problems.

- Different hardware models

  Whenever possible, run your multithreaded program on different hardware models with varying workload and stress levels. Different hardware models, especially multiprocessor systems, reveal a

variety of problems. If you are porting an application from another platform, verify that your application produces the same result on both platforms.

Testing is only one consideration for validating the correctness of a multithreaded program. Code inspections are critical and often the only way to determine potential problems. When inspecting the code for a multithreaded program, you need to continually ask this question, "If the code is interrupted at this point, what potential problems can occur?" Often, knowing the potential sources of problems helps you avoid them. Common problems are as follows:

- Hardware-related problems

  It is typically sufficient to test a single-threaded program on one hardware model and expect that it will behave identically on different hardware models. Unlike single-threaded programs, you cannot have the same expectation for a multithreaded program that runs on a hardware platform that supports kernel threads. On these platforms, each thread runs in a separately dispatched task. Depending on the processor speed, main memory sizes, storage capacity, and other hardware characteristics, timing problems or race conditions might occur when the multithreaded programs are called on different hardware. Multiprocessor systems further increase the likelihood of discovering potential problems. On multiprocessor systems, two different threads can run the same sequence of code at the same time.

- Java virtual machine implementation problems

  A problem for Java application developers is the implementation of the Java virtual machine (JVM). On some platforms, the JVM is single-threaded. On the i5/OS operating system, the JVM is multithreaded. Similar problems might occur with a Java application as with an application from another platform that supports user threads.

- Threads model problems

  If you are porting an application from another platform, you must know the threads model that the other platform supports. If the other platform supports a user threads model, you might encounter problems with the application on i5/OS because i5/OS supports a kernel threads model. A key difference between the two models is that the user threads model uses cooperative scheduling, whereas the kernel threads model uses preemptive scheduling. With a user threads model, only one thread in the application can be active at a given time. With a kernel threads model, the machine determines which threads are eligible to run. Race conditions, resource contention, and other problems might occur with kernel threads on i5/OS, because the application writer did not consider these potential problems when using user threads.

Performance considerations in multithreaded applications

There are several performance considerations for multithreaded jobs on the i5/OS operating system.

The concepts that are described here pertain to all programming languages. For details about how each language enables a concept, refer to the Language topic in the information center for that specific language.

Multithreaded server recommendations

When you write a typical server program to take advantage of threads, a common design is to have a single listener thread wait for client requests and to have multiple individual worker threads perform the operations requested by the clients.

Creating threads should not use many resources. However, because some servers require a faster response time to clients, some server application programs maintain a pool of worker threads that wait for work to avoid creating new threads.

Typically, the worker threads use some synchronization primitive to wait for client processing requests. Instead of creating a new thread to process each client request, the listener thread queues the client requests and signals the waiting worker threads. The signal action sometimes uses variables.

A server application is commonly considered trusted with the data that it serves to clients. Because the server is running multithreaded applications, you must consider some issues concerning the activities that the server performs:

- You must not call user application code from a multithreaded server. To run user application code safely from a multithreaded server, the user application code must follow the same strict rules that the original multithreaded server uses. These rules concern the actions that it can take and concern the application programming interfaces (APIs) that it can call.
- As you should with any other part of your application, you must evaluate the processing that is required to fulfill the client request for its thread safety.
- Processing on behalf of the client might affect process-level resources of the server in a way that is not desired by the server. For example, changing the coded character set identifier (CCSID) so that data representation is the same as that of the client also affects other threads in the job. CCSID is a job resource.
- A server can change the security information (user profile and group profile) of a worker thread to make it become the client that is being served. However, when the server does this, you need to consider what resources the threads share. The worker thread has access to all the already opened job level resources that more privileged users in the same job might have created or opened.

Job and thread priorities

An i5/OS thread competes with threads throughout the system for scheduling resources in addition to competing with other threads in the same job.

The system schedules processing resources by using a delay cost scheduler that is based on several delay cost curves (priority ranges).

On i5/OS, numerically lower priority values indicate higher priority with regard to scheduling. By adding its thread priority adjustment value to the job's priority, you can specify the priority of a thread. The default priority for a thread is no change from the process priority or a thread priority adjustment value of zero.

You can directly affect the application performance by assigning different thread priority adjustment values to the threads in your application.

Contention among threads

Contention occurs when one thread has to wait for another thread to finish using a resource.

Contention problems can occur if your application uses too few mutual exclusions (mutexes) to protect access to a large number of resources. A large number of threads that share a small number of resources can also cause contention between threads in your application.

Contention between threads over resources can cause context switches and paging. To reduce contention within your application, you should hold locks for the shortest amount of time and try to prevent a single lock from being used for two different or unrelated shared resources.

Threads that poll or spin to wait for resources prevent scaling. This form of contention can drastically affect the performance of your application. Polling or spinning can also adversely affect other threads or jobs, degrading system performance. Use condition variables, semaphores, mutexes, or other synchronization primitives so that threads never need to poll or spin to wait for resources to become available.

Effects of storage pool sizes on threaded applications

The storage pool size specified for a subsystem affects the performance and number of threads you can create.

If thread creation or processing is slow or fails consistently with an out-of-memory error, you might not have enough resources available in the storage pool to run all of your application threads in addition to the other jobs that are using the same storage pool. Increasing the storage pool size might eliminate these problems.

You can examine the storage pool sizes by using the Work with System Status (WRKSYSSTS) command. If you have too many active threads or threads that use too many system resources, you will see numerous page faults for the storage pools in which applications are running. Increasing the storage pool size might reduce the number of page faults.

Activity levels of storage pools

The activity level of a storage pool refers to the number of active threads within that storage pool.

A single job with 500 threads takes up the same number of activity level slots as 500 jobs with a single thread. If the activity level is too low for the number of threads and jobs that are active, your threads are paged out of main storage and marked as ineligible for a short time. This can drastically affect the performance of your application.

Performance and threaded applications

Job-level performance counters are updated by all the threads in a job.

The presence of multiple active threads can affect the accuracy of both general system counters and performance monitor specific transaction boundary counters. Data loss might occur as there is no automatic synchronization for these counters.

The initial thread performance information is a combination of thread and job-scoped data. Derived information that involves both thread and job-scoped data is not valid unless you total the thread-scoped data across all threads.

Examples: Threads

These examples are used throughout the Multithreaded application topic collection.

Use the following information as examples for your programs.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 72.

Pthread examples:	Java examples:
• "Example: Setting a thread attribute in a Pthread program" on page 13	• "Example: Setting a thread attribute in a Java program" on page 14
• "Example: Starting a thread in a Pthread program" on page 15	• "Example: Starting a thread using a Java program" on page 16
• "Example: Ending a thread in a Pthread program" on page 17	• "Example: Ending a thread using a Java program" on page 18
• "Example: Canceling a thread in a Pthread program" on page 20	• "Example: Canceling a thread using a Java program" on page 21
• "Example: Waiting for threads in a Pthread program" on page 24	• "Example: Suspending a thread using a Java program" on page 23
• "Example: Using mutexes in Pthread programs" on page 34	• "Example: Waiting for threads using a Java program" on page 26
• "Example: Using semaphores in Pthread programs to protect shared data" on page 38	• "Example: Using mutexes in a Java program" on page 36
• "Example: Using condition variables in Pthread programs" on page 40	• "Example: Using condition variables in Java programs" on page 42
• "Example: Space location locks in Pthread programs" on page 45	• "Example: Thread-specific data in a Java program" on page 53
• "Example: One-time initialization in Pthread programs" on page 48	
• "Example: Thread-specific data in a Pthread program" on page 51	
• "Example: Sample flight recorder output from a Pthread program" on page 63	
• "Example: Working with local SQL databases in multithreaded Pthread programs" on page 8	

Related reference

"Threads considerations for Java language" on page 59
Java threads operate on top of the i5/OS kernel threads model using the java.lang.Thread class. Each Java thread is one of the many tasks that run in the process.

Related information for Multithreaded applications

Product manuals, Web sites, and other information center topic collections contain information that relates to the Multithreaded applications topic collection. You can view or print any of the PDF files.

Manuals

• *WebSphere® Development Studio: ILE C/C++ Programmer's Guide, SC09-2712* PDF

• *WebSphere Development Studio: ILE C/C++ Language Reference, SC09-7852* PDF

• *ILE Concepts, SC41-5606* PDF

Web site

The Sun Java tutorial

Other information
• APIs
• Language

Code license and disclaimer information

IBM grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

SUBJECT TO ANY STATUTORY WARRANTIES WHICH CANNOT BE EXCLUDED, IBM, ITS PROGRAM DEVELOPERS AND SUPPLIERS MAKE NO WARRANTIES OR CONDITIONS EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, REGARDING THE PROGRAM OR TECHNICAL SUPPORT, IF ANY.

UNDER NO CIRCUMSTANCES IS IBM, ITS PROGRAM DEVELOPERS OR SUPPLIERS LIABLE FOR ANY OF THE FOLLOWING, EVEN IF INFORMED OF THEIR POSSIBILITY:

1. LOSS OF, OR DAMAGE TO, DATA;
2. DIRECT, SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES, OR FOR ANY ECONOMIC CONSEQUENTIAL DAMAGES; OR
3. LOST PROFITS, BUSINESS, REVENUE, GOODWILL, OR ANTICIPATED SAVINGS.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF DIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, SO SOME OR ALL OF THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

Appendix. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation

Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, IBM License Agreement for Machine Code, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

This Multithreaded applications publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i5/OS.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

| Advanced Function Printing
| AnyNet
| DB2
| i5/OS
| IBM
| IBM (logo)
| Integrated Language Environment
| Intelligent Printer Data Stream
| IPDS
| Open Class
| System/36
| System i
| WebSphere

| Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks
| of Adobe Systems Incorporated in the United States, and/or other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal Use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF

MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM ®

Printed in USA