libult Overview

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Table of Contents

1 Library Overview **3**
Purpose and Features 3
Embedding into a Program 3
Sample Programs 3
2 User Level Thread Usage Procedure **4**
Definition of User Level Threads 4
Basic Usage Procedure 4
3 User Level Thread Specifications **6**
State Transitions 6
Types of User Level Threads 7
Scheduling and Worker Thread Allocation Method 7
Used Resources 7
4 Synchronization Objects **8**
Overview 8
Data Structure 8
Waiting Queue Resource Pool 8
Mutex 9
Condition Variable 9
Reader/writer Lock 10
Semaphore 11
Queue 12
5 Precautions **14**
Interference Caused by Unfairness among Atomic Operation Cores 14
Thread Local Storage 14

1 Library Overview

Purpose and Features

libult (= User Level Threading library) is a library that provides lightweight user level threads and lightweight sync objects.

Embedding into a Program

The following files are required in order to use libult.

Filename	Description
ult.h	Header file
libSceUlt_stub.a	Stub library file
libSceUlt_stub_weak.a	Weak import stub library file

Copy sdk/target/sce_module/libult.suprx to a location that can be accessed from a program with app0:sce_module/libult.suprx.

Before calling the library functions, load PRX with the following code.

```
sceSysmoduleLoadModule( SCE_SYSMODULE_ULT );
```

After using the library, unload PRX with the following code.

```
sceSysmoduleUnloadModule( SCE_SYSMODULE_ULT );
```

Sample Programs

The following program is provided as a libult sample program for reference purposes.

samples/sample_code/system/api_libult/sample_ult/

This is a sample of the basic usage method of the user level threads and sync objects of libult.

2 User Level Thread Usage Procedure

Definition of User Level Threads

User level threads are threads that are implemented at the user level.

Their specifications are simple, and the threads can be manipulated without system calls, allowing thread creation and switching at a low cost compared to kernel provided threads.

Figure 1 Structure of User Level Threads

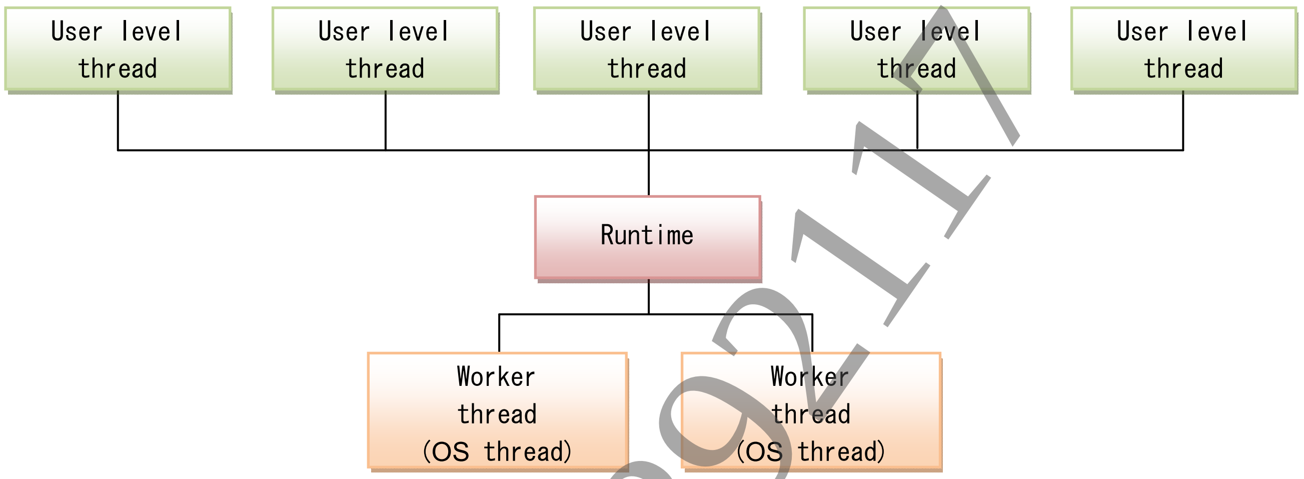


Figure 1 shows the structure of objects when user level threads are used. Runtime is an object that has the resources required for executing user level threads.

When an executable user level thread is registered to runtime, one of the worker threads belonging to runtime is assigned to and executes that user level. When the execution of the user level thread under execution is completed, or when the WAITING state is entered and the execution is paused, the worker thread executes the next executable user level thread.

Basic Usage Procedure

The basic usage procedure for user level threads is explained below. The basic processing flow is as follows.

(1) Create runtime

(2) Create user level threads

(3) Wait for user level thread completion

(4) Destroy runtime

In the following example, the explanation is based on the processing in sample_ulthread in the sample_ult sample.

(1) Create runtime

Execute `sceUltUlthreadRuntimeCreate()` to create runtime for executing the user level threads.

```
/* Maximum number of threads that can belong to runtime */
uint32_t maxNumUlthread = 16;
/* Number of worker threads */
uint32_t numWorkerThread = 3;
/* Get runtime work area size */
SceSize runtimeWorkAreaSize
```

```
       = sceUltUlthreadRuntimeGetWorkAreaSize(maxNumUlthread, numWorkerThread);
    /* Allocate work area */
    void *runtimeWorkArea = malloc(runtimeWorkAreaSize);
    /* Create runtime */
    SceUltUlthreadRuntime runtime;
    const char* runtime_name = "sample runtime"; /* Runtime name */
    SceUltUlthreadRuntimeOptParam *optParam = NULL; /* Use default option */
    sceUltUlthreadRuntimeCreate(&runtime,
                                runtime_name,
                                maxNumUlthread,
                                numWorkerThread,
                                runtimeWorkArea,
                                optParam);
```

(2) Create user level threads

Execute `sceUltUlthreadCreate()` to create user level threads and register them to runtime.

```
    /* Entry function of user level thread */
    int32_t sampleUlthreadEntry(uint32_t arg)
    {
       return SCE_OK;
    }

    SceUltUlthread ulthread;
    const *char ulthread_name = "sample ulthread"; /* Thread name */
    uint32_t arg = 0; /* Argument of entry function */
    /* Buffer for execution context of user level thread */
    void *contextBuffer = malloc(8192);
    uint64_t sizeContext = 8192; /* Context buffer size */
    SceUltUlthreadOptParam *threadOptParam = NULL; /* Use default option */
    sceUltUlthreadCreate(&ulthread,
                         ulthread_name,
                         sampleUlthreadEntry,
                         arg,
                         contextBuffer,
                         sizeContext,
                         &runtime,
                         threadOptParam);
```

(3) Wait for user level thread completion

Execute `sceUltUlthreadJoin()`, wait for user level thread completion, and free the context buffer.

```
    int32_t status; /* Variable for retrieving exit code of user level thread */
    sceUltUlthreadJoin(&ulthread, &status);
    /* Free context buffer */
    free(contextBuffer);
```

(4) Destroy Runtime

Execute `sceUltUlthreadRuntimeDestroy()` to destroy runtime.

```
    sceUltUlthreadRuntimeDestroy(&runtime);
    /* Free runtime work area */
    free(runtimeWorkArea);
```

3 User Level Thread Specifications

State Transitions

Figure 2 State Transitions of User Level Threads

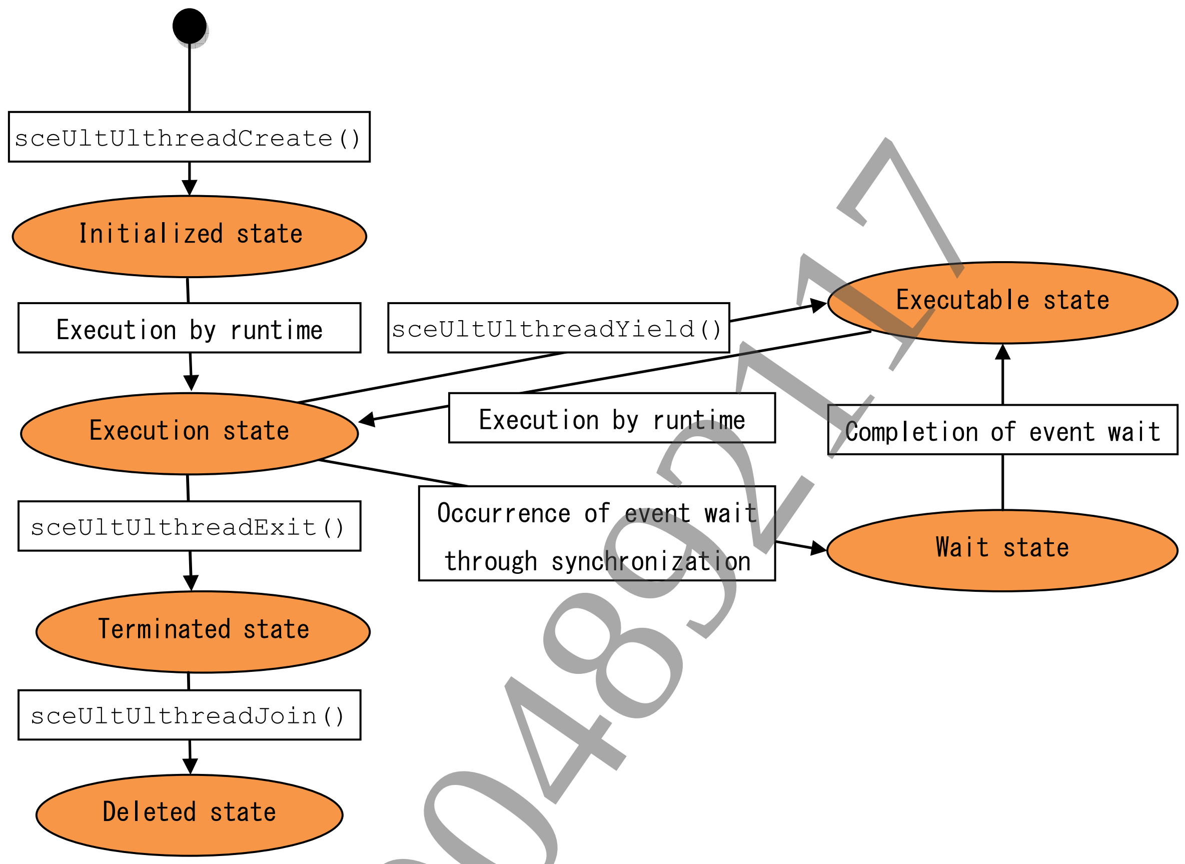


Figure 2 shows the user level thread states and their transitions.

Initialized state

This is the state immediately after creation. User level threads in this state are automatically executed by the worker threads of runtime.

Execution state

This is the state in which the user level thread is being executed by a worker thread.

Terminated state

This is the state in which execution is completed either as the result of return from the entry function or calling `sceUltUlthreadExit()`, and deletion by `sceUltUlthreadJoin()` is waited for.

Deleted state

This is the state in which `sceUltUlthreadJoin()` is completed and disconnection from runtime has been executed. When this state is entered, the memory area allocated to the user level thread can be freed.

Wait state

This is the state in which a specific event needs to be waited for during sync library use, and in which execution is temporarily stopped. Only user level threads that have a dedicated context area can enter this state.

Executable state

This is the state in which the event waited for is completed and execution becomes possible again. User level threads in this state are automatically executed by the worker threads of runtime.

Types of User Level Threads

User level threads are of two types, standard user level threads and one-shot threads.

A context area is allocated to standard user level threads when they are created, and during execution, they are executed using this area. On the other hand, in the case of one-shot threads, the area set for runtime during execution is temporarily allocated as a context area, and one-shot threads are executed using this area.

Unlike standard user level threads, one-shot threads do not have a dedicated context area, so once they have been executed, they cannot return to the wait state or executable state.

Table 1 User Level Thread Types

Type	Creation Method	Area Used during Execution	Function Differences
Standard user level thread	Specify a valid area for context area during creation	Area allocated for each user level thread	Following execution, the wait state and executable state can be entered. Synchronization functions that allow the wait state to be entered can be used.
One-shot thread	Specify NULL for context area during creation	Shared area allocated for each worker thread	Following execution, the wait state and executable state cannot be entered. Synchronization functions that allow the wait state to be entered cannot be used.

Note

One-shot threads can call synchronization functions that allow the wait state to be entered by specifying `SCE_ULT_ULTHREAD_ATTRIBUTE_FORCE_WAIT`. However, in this case, the worker threads are stopped when the wait state is entered, the worker threads do not execute threads during the wait state, even if there exists any executable thread. Please use this option with caution because there is a possibility to cause problems, such as deadlock, as the executable threads are not executed. Moreover, in the case that the wait state is entered by specifying this option, the thread switching cost is almost the same as that for the OS thread.

Scheduling and Worker Thread Allocation Method

If no user level thread is currently being executed, the worker thread selects one executable thread and starts executing it. At this time, the selected thread is the thread that first entered the initialized state or the executable state.

Used Resources

User level threads use only the memory area given in the argument. They do not use any other resources.

The user level thread runtime creates the number of OS threads specified in the argument at initialization and uses the memory area given in the argument.. No other resources are used.

4 Synchronization Objects

Overview

libult provides five types of synchronization objects: mutexes, condition variables, queues, semaphores, and reader/writer locks.

If operations can be executed immediately for a synchronization object, these operations are all executed with non-blocking processing, and if the operations cannot be executed immediately, the thread is stopped until it becomes executable.

All the synchronization objects can be used by both user level threads and OS threads.

Data Structure

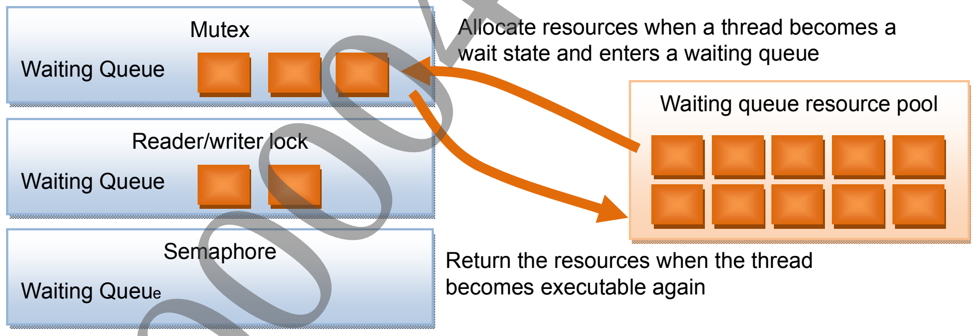
A synchronization object consists of the synchronization object itself and a waiting queue resource pool. In the case of a queue, a queue data resource pool is also included in the structure.

Waiting Queue Resource Pool

A waiting queue resource pool manages a memory area which is required when a thread enters a wait state.

When a thread that has operated a synchronization object enters a wait state, a temporary area for storing the thread information is allocated from a waiting queue resource pool connected to the synchronization object. This memory area is returned to the waiting queue resource pool when the thread becomes executable again.

Figure 3 Structure of Waiting Queue Resource Pool and Synchronization Object


Mutex
Waiting Queue
Reader/writer lock
Waiting Queue
Semaphore
Waiting Queue
Allocate resources when a thread becomes a wait state and enters a waiting queue
Waiting queue resource pool
Return the resources when the thread becomes executable again

If a waiting queue resource pool is not specified when a synchronization object being created, an operation having a possibility of stopping a thread cannot be executed for the object.

(1) Waiting queue resource pool creation

Allocate the work area required for creation and execute the create function.

```
SceUltWaitingQueueResourcePool waitingQueueResourcePool;

uint32_t numThreads = 16; /* Number of threads for which mutual exclusion control
is performed */
uint32_t numSyncObjects = 16; /* Number of synchronization objects which use this
waiting queue resource pool */
/* Allocate work area */
SceSize workAreaSize =
     sceUltWaitingQueueResourcePoolGetWorkAreaSize(numThreads,
                                                             numSyncObjects);
void *workArea = malloc(workAreaSize);

const char *name = "waiting queue"; /* Name */
SceUltWaitingQueueResourcePoolOptParam *optParam = NULL;/* Option (use default
value) */
/* Create waiting queue resource pool */
sceUltWaitingQueueResourcePoolCreate(&waitingQueueResourcePool, name,
                                     numThreads, numSyncObjects,
                                     workArea, optParam);
```

After the creation, specify this waiting queue resource pool when creating each synchronization object and use it.

(2) Waiting queue resource pool destruction

Following the destruction of all synchronization objects connected to the waiting queue resource pool, destroy the waiting queue resource pool.

sceUltWaitingQueueResourcePoolDestroy(&waitingQueueResourcePool);

Mutex

A mutex provides a lock mechanism for mutual exclusion control. A mutex can be unlocked only by the thread that locked it.

Use mutexes with the following procedure.

(1) Mutex creation

Allocate the work area required for creation and execute the create function.

```
SceUltMutex mutex;
const char *name = "mutex"; /* Mutex name */
SceUltMutexOptParam *optParam = NULL;/* Option (use default value) */
/* Create mutex */
sceUltMutexCreate(&mutex, name, &waitingQueueResourcePool, optParam);
```

(2) Lock

```
sceUltMutexLock(&mutex);
```

(3) Unlock

```
sceUltMutexUnlock(&mutex);
```

(4) Mutex destruction

Following mutex destruction, free the work area.

```
sceUltMutexDestroy(&mutex);
/* Free work area */
free(workArea);
```

Condition Variable

A condition variable provides functions to bind mutexes, pass locks, and perform signal notification and reception through indivisible operations.

Use condition variables with the following procedure.

(1) Condition variable creation

```
SceUltConditionVariable cv;

const char *name = "cv"; /* Name of condition variable */
/* Option (use default value) */
SceUltConditionVariableOptParam *optParam = NULL;
/* Create condition variable */
sceUltConditionVariableCreate(&cv, name, &mutex, optParam);
```

(2) Signal notification

```
sceUltMutexLock(&mutex);
…
sceUltConditionVariableSignal(&cv);
…
sceUltMutexUnlock(&mutex);
```

(3) Signal reception

During signal reception, execute reception wait after acquiring the lock of the mutex in question.

```
sceUltMutexLock(&mutex);
…
sceUltConditionVariableWait(&cv);
…
sceUltMutexUnlock(&mutex);
```

(4) Condition variable destruction

```
sceUltConditionVariableDestroy(&cv);
```

Reader/writer Lock

Reader/writer lock is a function to perform mutual exclusion control similar to mutex.

Unlike mutex, two types of locks can be used, reader lock and writer lock. Reader lock can be acquired by multiple threads simultaneously, while writer lock can be acquired by only one thread at a time. Reader lock and writer lock cannot be acquired at the same time.

Use reader/writer locks with the following procedure.

(1) Reader/writer lock creation

Allocate the work area required for creation and execute the create function.

```
SceUltReaderWriterLock rwlock;

const char *name = "rwlock"; /* Name of reader/writer lock */
/* Option (use default value) */
SceUltReaderWriterLockOptParam *optParam = NULL;
/* Create reader/writer lock */
sceUltReaderWriterLockCreate(&rwlock, name,
                             &waitingQueueResourcePool, optParam);
```

(2) Reader lock acquisition

```
sceUltReaderWriterLockLockRead(&rwlock);
```

(3) Reader lock release

```
sceUltReaderWriterLockUnlockRead(&rwlock);
```

(4) Writer lock acquisition

```
sceUltReaderWriterLockLockWrite(&rwlock);
```

(5) Writer lock release

```
sceUltReaderWriterLockUnlockWrite(&rwlock);
```

(6) Reader/writer lock destruction

Following reader/writer lock destruction, free the work area.

```
sceUltReaderWriterLockDestroy(&rwlock);
/* Free work area */
free(workArea);
```

Semaphore

Semaphores are used to control access to shared resources among threads.

Multiple resources can be acquired and released for a semaphore, and threads can be stopped and made to wait until the specified number of resources has been acquired.

Use semaphores with the following procedure.

(1) Semaphore creation

Allocate the work area required for the creation and execute the create function.

```
SceUltSemaphore semaphore;

const char *name = "semaphore"; /* Name of semaphore */
/* Option (use default value) */
SceUltSemaphoreOptParam *optParam = NULL;
uint32_t numInitialResources = 0; /* Number of initial resources*/
/* Create semaphore */
sceUltSemaphoreCreate(&semaphore, name,
                      &waitingQueueResourcePool, optParam);
```

(2) Resource acquisition

Specify the number of resources during resource acquisition. If the number of resources of the semaphore does not match the specified number of resources, the thread is stopped until the specified number of resources has been acquired.

```
uint32_t numResource = 2; /* Number of resources to be acquired */
sceUltSemaphoreAcquire(&semaphore, numResource);
```

(3) Resource release

```
uint32_t numResource = 1; /* Number of resources to be released */
sceUltSemaphoreRelease(&semaphore);
```

(4) Semaphore destruction

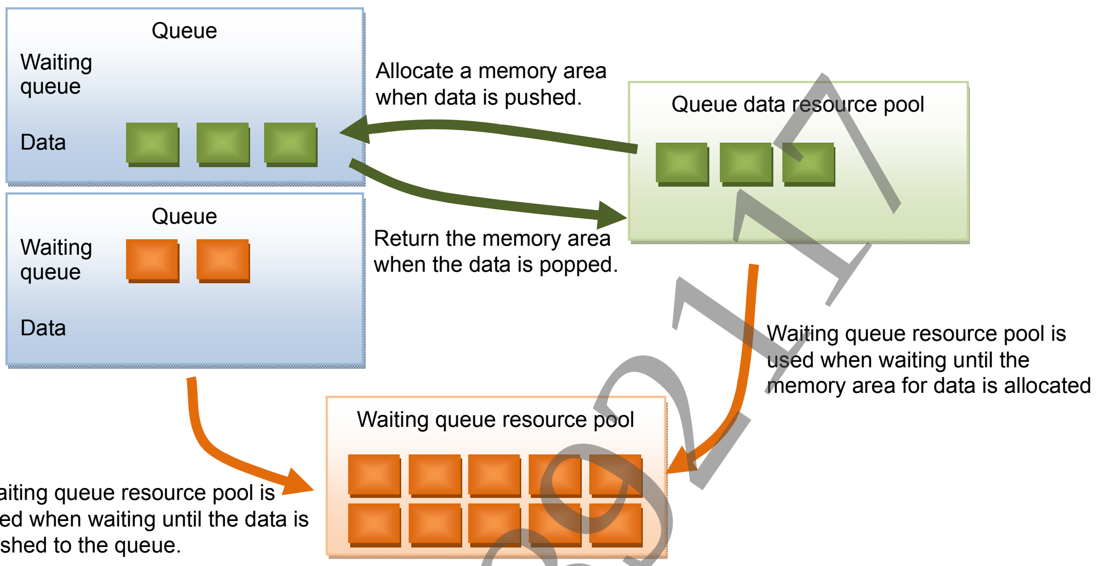
```
sceUltSemaphoreDestroy(&semaphore);
/* Free work area */
free(workArea);
```

Queue

A queue is a data container for transferring data between threads. The data can be retrieved from the queue in the order in which it was added to the queue.

A queue consists of the queue itself and a queue data resource pool which manages a memory area for data. When data is pushed to a queue, a memory area for the data is allocated from a queue data resource pool, and when the data is popped from the queue, the memory area is returned to the queue data resource pool.

Figure 4 Structure of Queue and Queue Data Resource Pool


Queue
Waiting queue
Data
Allocate a memory area when data is pushed.
Queue data resource pool
Queue
Waiting queue
Data
Return the memory area when the data is popped.
Waiting queue resource pool is used when waiting until the memory area for data is allocated
Waiting queue resource pool
Waiting queue resource pool is used when waiting until the data is pushed to the queue.

Use queues with the following procedure.

(1) Queue data resource pool creation

Allocate the work area required for creation and execute the create function.

```
SceUltQueueDataResourcePool queueDataResourcePool;

uint32_t queueDepth = 32; /* Depth of queue */
uint32_t dataSize = 16;    /* Size of 1 data */
uint32_t numQueue = 1; /* Number of queues which share a queue data resource pool
*/
SceUltWaitingQueueResourcePoolOptParam *resourcePoolOptParam = NULL; /* Option
(use default value) */
const char* resourcePoolName = "queueDataResourcePool"; /* Name of queue data
resource pool */
/* Allocate work area for a queue data resource pool */
uint32_t workAreaSize
= sceUltQueueDataResourcePoolGetWorkAreaSize(queueDepth, dataSize, numQueue)
void *queueDataResourcePoolBuffer = malloc(workAreaSize);

/* Create a queue data resource pool */
sceUltQueueDataResourcePoolCreate(&queueDataResourcePool,
                                  resourcePoolName,
                                  queueDepth, dataSize, numQueue,
                                  &waitingQueueResourcePool,
                                  queueDataResourcePoolBuffer,
                                  resourcePoolOptParam);
```

(2) Queue creation

Associate a queue with a queue data resource pool, and create the queue.

```
SceUltQueue queue;

const char *queueName = "queue"; /* Name of queue */
SceUltQueueOptParam *queueOptParam = NULL; /* Option (use default value) */

sceUltQueueCreate(&queue, queueName, dataSize, &queueDataResourcePool,
                  &waitingQueueResourcePool, queueOptParam);
```

(3) Data addition

When data is added to the queue, the amount of data corresponding to the data size specified during initialization from the pointer specified by the argument is copied to the queue.

```
void* data = …; /* Data of size specified by dataSize */
sceUltQueuePush(&queue, data);
```

(4) Data retrieval

When data is retrieved from the queue, the amount of data corresponding to the data size specified during initialization is copied from the queue to the pointer specified by the argument.

```
void* data = …;
/* Buffer for receiving data of the size specified by dataSize */
sceUltQueuePop(&queue, data);
```

(5) Queue destruction

```
sceUltQueueDestroy(&queue);
```

(6) Queue data resource pool destruction

Following the destruction of queue data resource pool, free the work area.

```
sceUltQueueDataResourcePoolDestroy(&queueDataResourcePool);
/* Free work area */
free(queueDataResourcePoolBuffer);
```

5 Precautions

Interference Caused by Unfairness among Atomic Operation Cores

Most functions of libult are implemented using ARM atomic operations.

Owing to the nature of the processor, if operations are executed continuously in relation to a particular libult object, such as repeated mutex locking and unlocking over a small interval of time when a busy loop occurs, only the operation of a particular thread may succeed and the operations of the other threads may be inhibited.

Thread Local Storage

When switching a user level thread, the OS thread that executes the user level thread may be changed. Thus, it is not guaranteed that a user level thread can refer to the same thread local storage at any time.