Kernel Overview

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Document serial number: 000004892117

Document serial number: 000004892117

Table of Contents

1 About This Document

Purpose

This document describes the features of the kernel. It provides the necessary information for writing a program using the kernel APIs (not described in the "Kernel Reference" document). It also describes some samples provided with the SDK as a tutorial.

Scope of Document

This document describes how to use the kernel APIs, but it does not provide information on all interfaces. It focuses on the usage description, so details of the interface specifications are only described where necessary. Refer to the "Kernel Reference" document for detailed specifications of each interface.

The reader is required to have general knowledge about the concepts and methods of programming, including processes, threads, and mutexes.

2 Kernel Introduction

About the Kernel

The kernel is the kernel system that runs on the PlayStation®Vita. The kernel uses a priority-based multithreaded scheduling mechanism, making it suitable for simple real-time processing. Semaphores, event flags and message boxes are used to provide features such as inter-thread synchronization and communication. The kernel also provides interrupt handler management features, memory pool management features, file interfacing features, timer management features, and dynamic link library features.

Software Hierarchical Structure of Kernel

System Memory Manager

The system memory manager manages the entire memory.

UID service

This service provides a unique identifier to the kernel resource.

Memory block service

This service allocates and frees continuous areas (memory blocks) of a virtual address in units of pages, at a relatively low frequency. This service is also used to allocate locations for loading program modules and to allocate a thread stack.

Exception Handler Manager

The exception handler manager provides features for registering, managing, and dispatching exception handlers.

Interrupt Handler Manager

The interrupt handler manager consists of routines that provide interrupt handler registration and management features, and an integrated interrupt handler for determining the cause of an interrupt and passing control to the appropriate handler. The PlayStation®Vita has multiple interrupt causes and provides functions for enabling and disabling interrupts for each cause by setting both a per-cause mask register and a master mask register. When an interrupt occurs, the interrupt handler manager reads the interrupt status register and the mask registers to determine the interrupt cause and calls the appropriate registered interrupt handler.

Multithread Manager

The multithread manager manages CPU resources, memory resources, and interrupt resources, and provides the following features to higher level software.

- Thread management
- Thread scheduling
- Inter-thread synchronization
- Callbacks
- Timers

IO/File Manager

The IO/File manager provides a device-independent file access API similar to that of `sceIoOpen()`, `sceIoClose()`, `sceIoRead()`, and `sceIoWrite()`. It also performs registration management for entry groups of these APIs, which are kept by device drivers and the filesystem.

Module Manager

The module manager manages the loading and unloading programs. It also provides the dynamic link libraries feature.

Process Manager

The process manager starts and ends processes and manages the process resources.

Kernel Library

This implements part of the kernel features in the user space.

Device Drivers, Filesystem and Kernel Services Group

Device Drivers

Device drivers are modules that use the features of the multithread manager to handle I/O devices. A device driver consists of the following three components.

- Interrupt processing routine (if needed)
- Device control thread (0 or more)
- Entry function group

The entry function group is called from a higher-level program. It is responsible for passing I/O requests to device control threads and for directly controlling devices. The entry function group is executed in the calling thread's context.

Filesystem

The filesystem is responsible for linking a general-purpose file interface to block devices. It also provides directory services.

Kernel Services Group

In addition to the above kernel modules, other kernel modules or non kernel mode modules provide individual services.

Kernel API Overview

The kernel APIs are abstracts of the kernel features mentioned above and are provided to the application developer as APIs. These APIs are thread-safe, and they consist of system calls provided by the kernel and various libraries executed by userland. More specifically, this refers to functions, macros and structures defined and declared with a target/include/kernel.h header file.

Limitation on Kernel Resources

A certain amount of each internal resource of kernel (memory, etc.) will be consumed when an application creates, for example, threads, synchronization objects and memory blocks.

Since the internal resources of kernel are not limitless, the maximum number of the objects that an application can create is determined as follows.

If an application attempts to create objects exceeding the maximum number, the API that creates such objects (`sceKernelCreateThread()` or other APIs) will return the `SCE_KERNEL_ERROR_UID_RL_OVERFLOW` error.

- Threads: 128
- The total number of the following synchronization objects: 2048
  - Event flags
  - Semaphores
  - Mutexes
  - Condition variables
  - Reader/writer locks
  - Lightweight mutexes
  - Lightweight condition variables
  - Simple events
  - Message pipes
  - Timers
- Memory blocks: 2048

Note that the above number also includes the number of objects created through such as libraries or middleware used by the application.

However, the above limitation does not apply to the objects created with Common Dialog.

3 Process Management

Overview

The kernel provides a process model. Resources between processes are protected by managing the resources, such as the process address space, real memory, and threads for each process.

Process Identifier

Each process has a unique process identifier (PID). A process identifier is a positive integer with a 32-bit length given by the kernel when a process is created.

Process Resources

Each process has a process address space and a user-main thread created when the process is generated. In addition, a process can have the desired number of user threads. (The upper limit of the number of each resource depends on the kernel implementation and physical memory size.) A process-local synchronization primitive is included in the process resources.

Each process resource has an identifier. These identifiers are locally unique within the process.

Creating a Process

A game process cannot create a subprocess.

Terminating a Process

Spontaneous process termination is prohibited by the TRC (Technical Requirements Checklist). For details, refer to the "Application Termination Processing" chapter in the "Programming Startup Guide" document.

Resource Collection When Terminating a Process

When process resources cannot be collected and remain after terminating a process, the kernel collects those resources. The following are the detailed actions.

Threads

Regardless of the status of the thread, the resources are collected. If the thread is being executed, it is forcibly terminated without being allowed to be executed to completion.

Synchronization primitives

The resources of a synchronization primitive are collected when the process is terminated.

Memory mapped to a process address space

All resources of the memory mapped to a process address space are freed.

Modules

All loaded modules are unloaded.

Priority and Stack Size of User Main Thread

The priority of the user main thread executing `main()` is initially set with `SCE_KERNEL_DEFAULT_PRIORITY_USER` (in the case of a game application, this will be internally converted to 160) and the stack size is initially set with `SCE_KERNEL_THREAD_STACK_SIZE_DEFAULT_USER_MAIN(256*1024)`. The initial setting values can be changed by defining the following global variables in the main module.

```
#include <kernel.h>
extern const char   sceUserMainThreadName[]     __attribute__((weak));
extern int          sceUserMainThreadPriority   __attribute__((weak));
extern unsigned int sceUserMainThreadStackSize __attribute__((weak));
```

- `sceUserMainThreadName`
  Specify a string with the length in `SCE_UID_NAMELEN` as the maximum length for the name.
- `sceUserMainThreadPriority`
  Specify the scheduling priority.
- `sceUserMainThreadStackSize`
  Specify the stack size by the number of bytes.

In the following example, the scheduling priority is set to 128 and the stack size is set to 512*1024[B].

```
#include <kernel.h>
const char   sceUserMainThreadName[]     = "sample_main_thr";
int          sceUserMainThreadPriority   = 128;
unsigned int sceUserMainThreadStackSize = 512 * 1024;

int main(void)
{
    /* Main module processing */
```

Process Time

The elapsed time after starting up a process can be obtained in a microsecond unit as a process time by executing `sceKernelGetProcessTime()`. The process time is 64 bits wide and serves as a counter that can only be referenced. An application cannot stop the counter or set a count value. In addition, the counter stops counting while a process is suspended.

The process time can be used as a common time that can be referenced at any time by all threads in a process.

4 Module (PRX) Management

Overview

The module manager manages loading and unloading of programs stored in the file system to the memory and the dynamic link libraries ("libraries"). Programs are managed in units called modules and can be loaded to the memory or unloaded from the memory as desired. At this time, the load destination address of a module is assigned dynamically by the module manager. Multiple libraries can be declared within a module and exported to another module in units of libraries. Registration of libraries and dynamic linking of functions and variables are performed automatically when a module is loaded.

PlayStation®Vita uses the relocatable object file format called PRX, and hereafter in this document, the modules placed in the memory and PRX format files are collectively referred to as "PRX".

Note
In this SDK release, only one library can be exported per module.
For the build method employed for PRX files, refer to the "PRX Programming Overview" document.

PRX Identifier

A unique `SceUID` type value is allocated within a process as the PRX identifier to the PRX loaded in the memory. To unload a given PRX, specify that PRX through its identifier.

PRX Name and Attribute

A name consisting of up to 26 characters, not including the termination character "\0", can be assigned to a PRX. The `SCE_MODULE_INFO` macro is used to set the name. The set name can be checked via a debugger. In addition to the name, any two `SceUInt8` type values and attributes can be specified with the `SCE_MODULE_INFO` macro.

The value that can be set as an attribute is as follows.

`SCE_MODULE_ATTR_NONE` : No attribute specified

Note
In this SDK release, always specify `SCE_MODULE_ATTR_NONE` as the attribute value.

PRX Load and Start

Dynamic program loading is executed through the load processing, which places the program stored in the file system on the memory, the export of the libraries, and the start processing, which performs dynamic link processing. PlayStation®Vita provides `sceKernelLoadStartModule()` to simultaneously perform the load processing and start processing.

```
#include <kernel.h>

static SceUID load_prx(const char *path)
{
    return sceKernelLoadStartModule(path, 0, 0, 0, NULL, NULL);
}
```

Note
In this SDK release, no API for separately performing the PRX load and start processing is provided.

PRX Stop and Unload

PRX unloading is realized through the stop processing, which performs library deletion and unlink processing, and the unload processing, which performs memory release. PlayStation®Vita provides `sceKernelStopUnloadModule()` to simultaneously perform the stop and unload processing.

```
#include <kernel.h>

static int unload_prx(SceUID prx_id)
{
    return sceKernelStopUnloadModule(prx_id, 0, 0, 0, NULL, NULL);
}
```

Note
In this SDK release, no API for separately performing the PRX stop and unload processing is provided.

Entry Functions

Functions for start and stop operations, and a function called at process end can be defined for each PRX (entry functions).

`int module_start(SceSize, const void*)`	:	Name of function called during PRX start processing
`int module_stop(SceSize, const void*)`	:	Name of function called during PRX stop processing
`void module_exit(void)`	:	Name of function called at process end

Either of the entry functions is called if defined in the PRX

Note
In this SDK release, entry function names cannot be changed.

`module_start` Function

`module_start()` is a function that is called from a thread whose priority and stack size have been initially set in `SCE_KERNEL_DEFAULT_PRIORITY_USER` and `SCE_KERNEL_THREAD_STACK_SIZE_DEFAULT_USER_MAIN`, respectively, during the PRX start processing. The argument block specified with the *`args`* and *`argp`* arguments of `sceKernelLoadStartModule()` is passed to `module_start()` after it is copied to the thread's stack.

The values that can be specified as return values of `module_start()` are as follows.

`SCE_KERNEL_START_SUCCESS`	:	Start was successful (resident)
`SCE_KERNEL_START_NO_RESIDENT`	:	Start was successful (non-resident)
`SCE_KERNEL_START_FAILED`	:	Start failed (non-resident)

If `SCE_KERNEL_START_NO_RESIDENT` is returned, PRX does not reside in the memory and is automatically unloaded instead (`module_stop()` call is not performed). At this time, the return value of `sceKernelLoadStartModule()` is `SCE_OK`. If `SCE_KERNEL_START_FAILED` is returned, the start processing is considered to have failed, and the processing of `sceKernelLoadStartModule()` fails. At this time, `SCE_KERNEL_ERROR_MODULEMGR_START_FAILED` is returned as the return value.

`module_stop` Function

`module_stop()` is a function that is called from a thread whose priority and stack size have been initially set in `SCE_KERNEL_DEFAULT_PRIORITY_USER` and `SCE_KERNEL_THREAD_STACK_SIZE_DEFAULT_USER_MAIN`, respectively, during the PRX stop processing. The argument block specified with the *`args`* and *`argp`* arguments of `sceKernelStopUnloadModule()` is passed to `module_stop()` after it is copied to the thread's stack.

The values that can be specified as return values of `module_stop()` are as follows.

`SCE_KERNEL_STOP_SUCCESS` : Stop was successful
`SCE_KERNEL_STOP_CANCEL` : Did not stop

If `SCE_KERNEL_STOP_CANCEL` was returned, the stop processing was interrupted and the processing of `sceKernelStopUnloadModule()` failed. At this time, `SCE_KERNEL_ERROR_MODULEMGR_STOP_FAIL` is returned as the return value.

`module_exit` Function

`module_exit()` is called from the context of the thread that called the process end function. At this time, `module_exit()` call is done in the reverse order in which PRX loading was performed. `module_exit()` call applies only to the PRX not unloaded at process end.

> **Note**
> `module_exit()` is called only in the **Development Mode** of the Development Kit. For details, refer to the "Application Termination Processing" chapter in the "Programming Startup Guide" document.

Stub Files and Loading Sequence

In PlayStation®Vita, the libraries exported from PRX can be used by simply linking statically stub files (*_stub.a). In the SDK, the following two stub files are provided for each system library.

`<libname>_stub.a` : Stub file for safe import (safe link)
`<libname>_stub_weak.a` : Stub file for loose import (loose link)

If a stub file for safe import (safe link) has been statically linked, `SCE_KERNEL_ERROR_MODULEMGR_NO_LIB` error occurs if it is loaded with `sceKernelLoadStartModule()` without previously loading the PRX that exported that library.

If a stub file for loose import (loose link) has been statically linked, loading by `sceKernelLoadStartModule()` is successful even without previously loading the PRX that exported that library. However, the functions and variables of the library that was loose imported (loose linked) cannot be accessed as is. Load the PRX in question explicitly before accessing these functions and variables. When a function that is not dynamically linked is called, a prefetch abort exception occurs for address 0. (In the case of variable access, a data abort exception occurs for address 0).

Note that using stub files for loose import (loose link) without careful consideration leads to more complex dependency relationships among PRX's.

> **Note**
> In PlayStation®Vita, the stub file for safe import (*_stub.a) statically linked to the main module (*.self) is considered to have been statically linked to the stub file for loose import automatically by the system.

Behavior during Call of Function that Is Not Dynamically Linked

In PlayStation®Vita, calling a function that is not dynamically linked causes a prefetch abort exception for address 0 and also the following debug message to be displayed on the Neighborhood for PlayStation®Vita console output. The following is a display example.

```
=== Caller Module Info (that caused PABT) ==============
Module Name    : SamplePrx
Path           : host0:sample_prx.self
Stub Code Addr : 0x8100061c (seg:offset 0:0x0000061c)
Library Name   : SamplePrxSub (ver:1)
Function NID   : 0x75c7f0d7
========================================================
```

The above display example shows that calling a function exported from the SamplePrxSub library (function `NID`: 0x75c7f0d7) in the SamplePrx program (path name: host0:sample_prx.self) was attempted, but because this function was not dynamically linked, a prefetch abort exception occurred (0x0000061c the stub function's address is 0x8100061c, the segment number containing the stub function is 0, and the offset from the beginning of that segment is 0x0000061c).

The function `NID` is a value that is statically assigned by a toolchain in relation to a function exported in a library so that it is unique within that library. The function name can be determined by searching for the symbol name with the same value as the function `NID` from the program file before the symbol information has been stripped.

5 Thread Management

This section describes the multithread management features of the kernel.

Thread Manager Overview

The thread manager provides the following features.

- Multithread management (creating, starting, stopping, deleting...)
- Synchronization mechanism
- Communication between threads
- Communication between processes
- Timers

Threads

Threads serve as the logical smallest executable unit of programs, as seen from the perspective of parallel processing. The processing performed by an application program is divided into multiple threads, and these threads are run simultaneously and in parallel.

At least one thread exists in a process.

Each thread has the following information internally. All such information is collectively referred to as thread context.

- Contents of the CPU register set
- Stack
- Thread local storage (TLS)

The user mode stack and user mode TLS are obtained from the application's memory budget at the time of thread creation. Therefore, it is necessary to consider the size as part of memory strategy of the application. (For the specific size, refer to the "Kernel Reference" document.)

Other context information is obtained from a memory budget reserved by the system beforehand, and thus the application needs not be aware of the size.

Threads can be executed simultaneously up to the maximum usable number of CPUs. Which thread to run on which CPU is determined by a scheduler. Elements that have an influence on the scheduling of a thread are "state", "priority" and "CPU affinity mask". Descriptions on each element are provided later in this document.

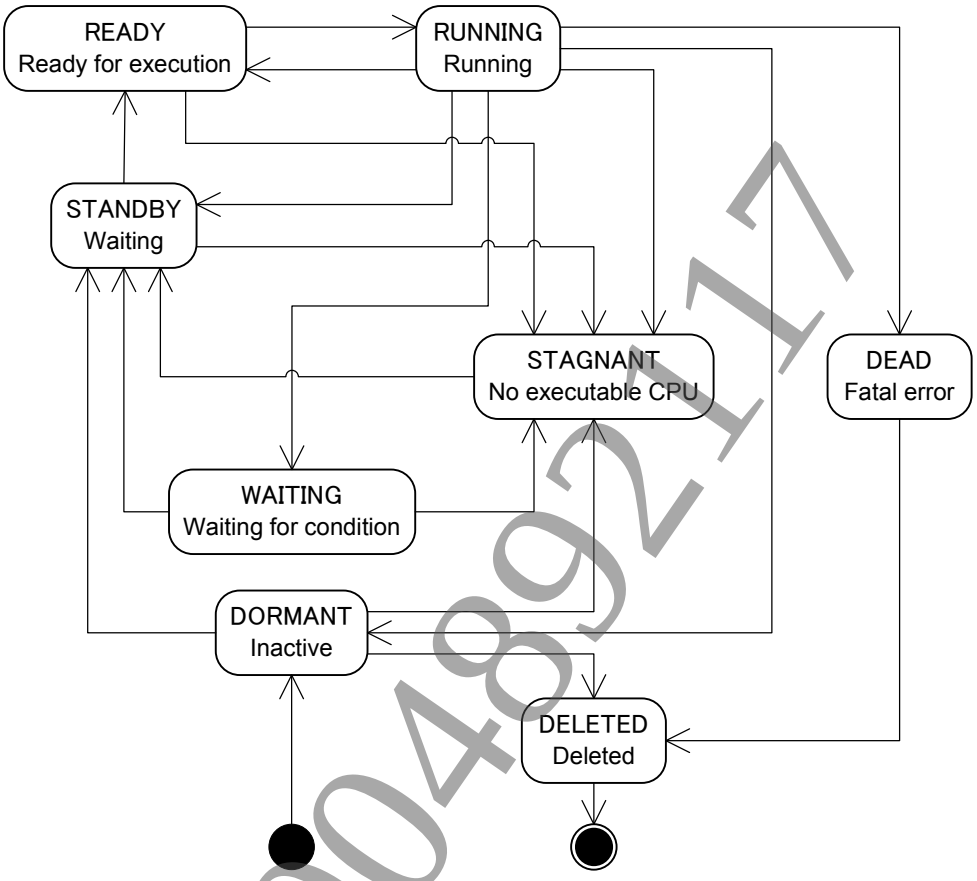
Each thread has a unique thread identifier (UID) in a process to which the thread belongs.

The thread identifier can be obtained as a return value of `sceKernelCreateThread()` when a thread is created. It is also possible to obtain the identifier by calling `sceKernelGetThreadId()` from the thread.

Thread States

Threads managed by the kernel have the following eight states. The following is the state transition diagram for threads and a description of the various states.

Figure 1 State Transition Diagram for Threads



DORMANT

The thread has been created but has not been activated yet, or the thread has been terminated.

This is the first state of a thread created with `sceKernelCreateThread()`.

STANDBY

The conditions for executing the thread are satisfied and the CPU to be executed is determined. (From the user's standpoint, there is virtually no difference when compared to READY state. These are separated because it is useful to differentiate the states during debugging.)

READY

The conditions for executing the thread on a specific CPU are satisfied, but either the thread that is running on that CPU has a higher priority than that of the waiting thread, or the priority levels are the same and, therefore, the thread remains in wait state and is not executed.

RUNNING

The thread is being executed on a CPU.

WAITING

The thread is waiting for a synchronization object or timer. It is not possible to execute the thread until the waiting state is released.

STAGNANT

The CPU to execute the thread cannot be determined because the usable CPUs on the system are not included in the CPU affinity mask of the thread.

The thread can be restored by changing the usable CPUs or the CPU affinity mask.

*Normally, an application needs not be aware of this state.

DELETED

The thread has been deleted, but the minimum information remains on the system because it is being referenced by the debugger or other tool.

DEAD

Execution of the thread has been stopped because of an unrecoverable error.

Thread Priority

Each thread has own priority. The range of the priority that can be set to a thread in a process is from `SCE_KERNEL_HIGHEST_PRIORITY_USER(=64)` to `SCE_KERNEL_LOWEST_PRIORITY_USER(=191)`. The lower number indicates the higher priority.

* The thread of the system may use the higher or lower priority than that in this range.

When a thread enters READY state, the thread is scheduled and executed according to the priority as a rule. (The thread transitions to RUNNING state.) A CPU execution right of a thread in RUNNING state will not be preempted by a thread having the same or lower priority.

The scheduler manages the thread's queue (ready queue) by priority. The ready queue is a queue to place the threads that have satisfied the conditions for execution (in READY state) in order of priority. All threads in READY state are connected to this queue.

The ready queue is divided into the following two parts: an individual part for each CPU (priority 64 to 127) and a shared part used by all CPUs (priority 128 to 191).

`SCE_KERNEL_DEFAULT_PRIORITY_USER` (=0x10000100) can be used for `sceKernelCreateThread()` and `sceKernelChangeThreadPriority()` as a priority to be specified. In the system, this value will be internally converted to the default priority which is set for each type of process the thread belongs to. If the process is used for a game application, the value is always set to 160. It is also possible to specify `SCE_KERNEL_DEFAULT_PRIORITY_USER` with from -31 to +32 offset value.

CPU Affinity Mask of Threads

Each thread has own CPU affinity mask. A CPU affinity mask is a bit mask used to specify an executable CPU for a thread. The thread is executed only on a CPU included in the specified CPU affinity mask. Furthermore, a CPU affinity mask is also set for a process. A CPU which is not included in a CPU affinity mask of a process the thread belongs to cannot be specified for the thread's CPU affinity mask.

Macros that can be specified as a CPU affinity mask are as follows.

- `SCE_KERNEL_CPU_MASK_USER_0`
- `SCE_KERNEL_CPU_MASK_USER_1`
- `SCE_KERNEL_CPU_MASK_USER_2`

The following macro can be used to specify at one time all the CPUs that can be used.

- `SCE_KERNEL_CPU_MASK_USER_ALL`

There exists a certain limit to the combination of a thread priority that can be specified and a CPU affinity mask.

For the details on the limit, refer to the "Thread Scheduling" section described below.

It is also possible to use `SCE_KERNEL_THREAD_CPU_AFFINITY_MASK_DEFAULT` (=0) for `sceKernelCreateThread()` and `sceKernelChangeThreadCpuAffinityMask()` as a CPU affinity mask to be specified. In the system, this value will be internally converted to the default CPU affinity mask which is set for each process the thread belongs to. If the process is used for a game application, all available CPUs can be indicated by that value.

Thread Scheduling

Overview of a Scheduler

Thread scheduling determines which thread to run on which CPU at a certain point among executable threads in the system (that is, threads in RUNNING, READY or STANDBY state).

The kernel implements a real time based multi-core thread scheduler.

The characteristics of the kernel scheduler are as follows.

Real time based and fixed priority

As long as a thread with higher priority is running on a CPU, a thread with lower priority will not be executed on the same CPU. Also, the priority of each thread is not changed without an explicit operation.

Not time sharing

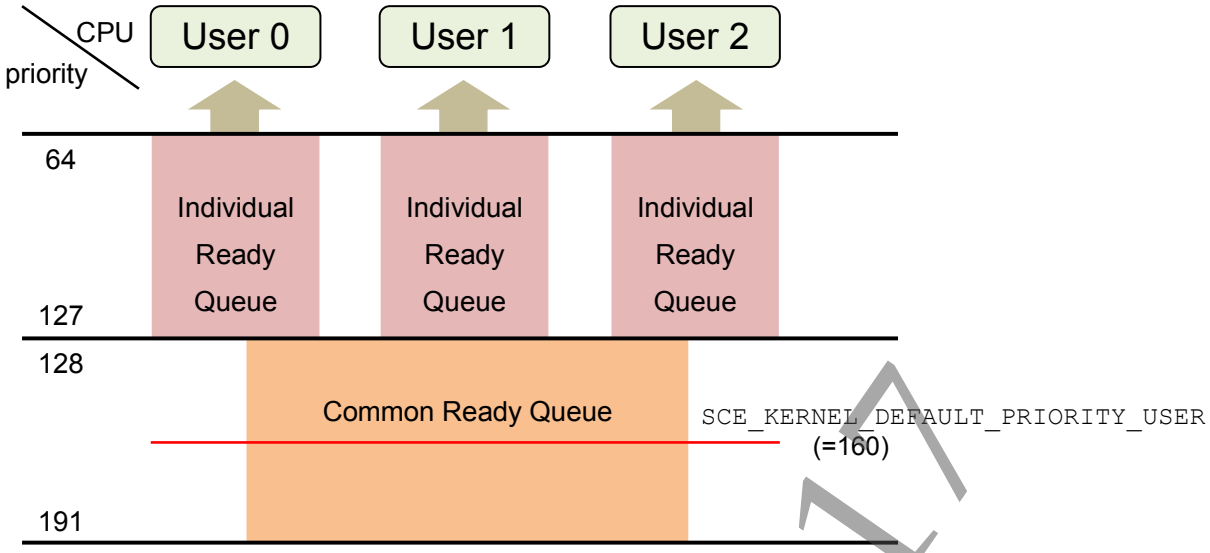
A CPU execution right is not automatically transferred from a thread which is being executed to another thread with the same priority.

Therefore, it is necessary to design an application so that each thread appropriately transfers the execution right to another thread by using the synchronization mechanism provided by the kernel.

Combined use of individual ready queue and common ready queue

The kernel scheduler divides the individual ready queue of each CPU and the common ready queue shared by all CPUs according to the range of the priority and uses them in combination. The image of the structure is shown below.

Figure 2 Image of Queue Structure of Scheduler



As shown in Figure 2, an individual ready queue and the common ready queue manage a thread with high priority and a thread with low priority respectively. When scheduling is performed on each CPU, the thread with the highest priority is searched from each CPU's individual ready queue first. If a thread does not exist in the individual ready queue, the thread with the highest priority is searched from the common ready queue. If a thread does not exist in both queues, the CPU enters an idle state and switches to the power saving mode.

An application can determine which ready queue to use (or both queues in combination) by adjusting the priority of each thread.

Individual Ready Queue

- Cost for scheduling processing is low.
- An optimum performance can be achieved by minutely controlling the CPU on which a thread is executed.
- Note that the priority may be reversed because it is not necessarily true that a thread with the second or lower priority is given an execution right in order of priority depending on the setting of CPU affinity mask.

For example, among the following three executable threads, the threads A and C are executed on the CPU USER 0 and USER 1 respectively, whereas the thread B is forced to wait being left in the READY state regardless of the priority A>B>C.

Thread A: Priority = 64, CPU affinity mask = USER_0

Thread B: Priority = 80, CPU affinity mask = USER_0

Thread C: Priority = 96, CPU affinity mask = USER_1

Common Ready Queue

- Load distribution among threads is automatically performed by the scheduler. Therefore, applications do not need to manage it.
- In the range of the common ready queue priority, it is guaranteed that the CPU execution right is always allocated to a thread in order of priority. (The priority is never reversed.)
- Cost for scheduling processing is high.

If the default value `SCE_KERNEL_DEFAULT_PRIORITY_USER` and `SCE_KERNEL_THREAD_CPU_AFFINITY_MASK_DEFAULT` are specified for the thread priority and the CPU affinity mask respectively, the thread is placed in the common ready queue. If you want to adjust the priority among the threads in the common ready queue using the default setting, specify an offset value from -32 to +31 for `SCE_KERNEL_DEFAULT_PRIORITY_USER`.

Notes on Priority and CPU Affinity Mask Combinations

Note that in thread scheduling, since each CPU always searches for and executes threads with high priority from their own individual ready queues, when looking at the overall system, threads are not always executed in order of priority.

The only guarantee is that one of the threads with the highest priorities in the overall system will be executed in one of the CPUs.

For example, a priority inverted situation may occur where even if a thread with low priority is being executed in CPU 1, a thread with high priority is waiting in a READY state in CPU 2.

In addition, once a thread has entered the individual ready queue of a CPU (or the common ready queue), in principle it will not move to another queue unless the scheduling has been re-performed for that thread.

Since behavior such as priority inversion that applications cannot easily predict can often cause problems, in order to prevent these kinds of phenomena from occurring, setting the following thread priority and CPU affinity mask combinations **is strongly recommended**.

- When setting an individual ready queue priority for a thread
  Specify only one CPU for a CPU affinity mask and explicitly "attach" the thread to a specific CPU.
- When setting a common ready queue priority for a thread
  - Specify `SCE_KERNEL_CPU_MASK_USER_ALL` for a CPU affinity mask.
  - The scheduler has sole responsibility for determining which CPU to use to execute a thread.

One the other hand, if a combination other than those recommended is used, the scheduling determinacy will decrease, and you will need to fully prepare for the occurrence of thread priority inversion described above and behavior that is different for each application execution.

Situations such as the following are strongly linked to the occurrence of bugs that have low recurrence and are difficult to debug.

Typical examples of occurrences are shown in the following.

Priority inversion example:

(1) Threads with a priority of 96 are running in CPUs 0, 1, and 2.
(2) Thread A with a priority of 97 and CPU affinity mask 0x3 (= CPU0 or 1) is scheduled for the individual ready queue of CPU 0.
(3) Thread B with a priority of 98 and CPU affinity mask 0x3 (= CPU 0 or 1) is scheduled for the individual ready queue of CPU 1.
(4) The thread running in CPU 1 terminates.
(5) Thread B is executed since it has the highest priority in the individual ready queue of CPU 1.

(6) Thread A has a higher priority than thread B and is in a state where it can be executed by CPU 1, but in this case thread B is executed.

In this case, if thread A were to be scheduled for CPU 1 in step (2), the priority inversion would not occur, therefore it can also be said that there is possibility of the behavior differing for every application execution.

Another such example is shown in the following.

Example of the behavior differing for every application execution:

(1) A thread with a priority of 96 is running in CPU 0, and CPUs 1 and 2 are idling.

(2) Thread C with a priority of 160 and CPU affinity mask 0x6 (= CPU 1 or 2) is scheduled for the common ready queue.
Here, since thread A can be executed in either CPU 1 or CPU 2, it will be executed in one of these CPUs. However, it is not possible to predict which one will be selected.

(3) Thread D with a priority of 160 and CPU affinity mask 0x5 (= CPU 0 or 2) is scheduled for the common ready queue.
If thread C is executed in CPU 1: thread D will be executed in CPU 2.
If thread C is executed in CPU 2: thread D does not have an executable CPU, so it will wait in a READY state.

Threads C and D both have the same priority in the common ready queue, but depending on the results of which CPU executes thread C in (2), two types of behavior are possible for (3): a case where both threads C and D are executed at the same time, or a case where only one is executed.

There is also a possibility of bias making one of these cases easier to occur, and if there is a problem in the behavior for the case which is less likely to occur, it will tend to cause bugs that are difficult to reproduce and debug.

Trigger for Scheduling

The following operations and events will be the trigger to perform scheduling on a CPU.

- When a thread is started up
- When a thread being executed is terminated
- When a thread being executed calls an API on its own to transition to WAITING state (waiting for a synchronization object, `sceKernelDelayThread()`, etc.)
- When waking up a thread that has been waiting for a synchronization object
- When a thread that has been waiting for a synchronization object is woken up by an interrupt processing
- When a thread moves between CPUs by, for example, changing the CPU affinity mask
- When the process to which a thread being executed belongs is suspended
- When an exception occurs for a thread being executed

With these operations, if a thread being executed on a CPU releases the CPU execution right on its own, or if a thread becomes executable on a CPU and the thread's priority is higher than that of the thread currently being executed on the CPU, scheduling processing will be performed on the CPU.

Determining the CPU to Execute a Thread

To which CPU's ready queue a new executable thread is connected is determined using the following conditions in order. For a thread for which a common queue priority is specified, the same procedure will be applied to determine a CPU to execute the thread. The following conditions are used as tips to determine on which CPU scheduling is performed.

(1) Confirm CPU affinity mask
(2) CPU currently executing an idle thread
(3) CPU currently executing a thread with a lower priority
(4) CPU currently executing a thread that belongs to the same process as that of the new thread
(5) CPU that previously executed a new thread
(6) If there is more than one CPU left, randomly select the CPU from the remaining candidates.
(7) After the CPU has been determined, place the thread in STANDBY state on that CPU. (In the case of a thread having a common ready queue priority, directly put the thread in READY state and connect to the ready queue.)

Thread Life Cycle

In general, a thread's life cycle is as follows.

Creating a Thread

When `sceKernelCreateThread()` is executed, the thread manager creates a thread and allocates the necessary resources. The thread is in DORMANT state at this point and is not subject to scheduling, so it is not actually executed by the CPU.

Starting a Thread

When `sceKernelStartThread()` is executed for the thread in DORMANT state, the thread is started and scheduled. Scheduling determines the CPU to execute the thread, thereby transitioning the thread to READY state. (In some cases, a thread may go through STANDBY state before transitioning to READY state.)

When a thread becomes the thread with the highest priority in the individual ready queue or common ready queue of each CPU, the thread enters RUNNING state and is executed.

A Thread in Waiting State

If a thread executes a wait function and transitions to WAITING state on its own, then the thread loses a CPU execution right. A thread in WAITING state waits until waiting conditions are satisfied by another thread or an interrupt processing. Once the waiting conditions are satisfied, the thread will wake up and be subject to scheduling again.

Preempting a Thread

When a thread enters READY state on a CPU after the thread becomes available for execution, if the thread's priority is higher than that of a thread being executed on the CPU, the CPU execution right will be taken away from the thread originally being executed. Then the thread with higher priority will be executed.

Stopping a Thread

When a thread in RUNNING state calls `sceKernelExitThread()` on its own or returns from the entry function, the thread stops and returns to DORMANT state.

Deleting a Thread

When a thread in RUNNING state calls `sceKernelExitDeleteThread()` on its own or another thread calls `sceKernelDeleteThread()` for a thread in DORMANT state, the thread is deleted from the system. However, threads that are being referenced by the debugger or other tools are not immediately deleted but remain in the system in DELETED state. Threads in DELETED state are deleted from the system when all references are gone.

Synchronization Mechanism

Synchronization Mechanism Overview

The thread manager provides several synchronization objects to perform synchronization between threads.

When a competitive access to a specific hardware resource or data in a memory occurs between multiple threads, exclusive control must be performed appropriately by using the synchronization mechanism provided by the kernel.

The synchronization mechanism is also used to wait for an event generated on another thread (or on the system).

While waiting for a synchronization object, a thread transitions to WAITAING state. If each thread appropriately enters WAITING state, the thread manager can put each CPU in the power saving mode as described before. For a thread, the so-called busy-wait polling can be used as a method to wait for an event other than the method using the synchronization mechanism. However, it is not possible to reduce the power consumption if using the busy-wait polling.

Since PlayStation®Vita is a mobile device, it is vital to reduce the wasted power consumption. Therefore, writing a program to perform busy-wait polling is not recommended. Control the threads by appropriately using the synchronization mechanism.

*** For the above reasons, kernel does not provide any API to perform basic yield operation.**

Each synchronization object has a unique thread identifier in a process (except for the thread delay).

Synchronization objects provided by the kernel are as follows.

- Threads
- Thread delay
- Event flags
- Semaphores
- Mutexes
- Condition variables
- Reader/writer locks
- Lightweight mutexes
- Lightweight condition variables
- Event objects
  - Simple events
  - Message pipes
  - Timers

Method for Using Synchronization Objects

Each synchronization object has different operations, but there are common usage methods for all the synchronization objects. First, in this section, the common methods are explained. For details on each object, refer to the explanation provided later as well as the section of each API in the "Kernel Reference" document.

Creating Synchronization Objects

The create API (`sceKernelCreate***()`) provided for each object is used for this operation. A unique object identifier (UID) returns from the create API and is used for the following operations and debugging.

It is possible to name an object with the length of up to 32-byte. When communication between processes is not performed, an object's name is not necessarily unique since the name is solely used for identification by an operator during debugging.

In addition, attributes can be specified for synchronization objects. There are two types of attributes: common attributes, which can be specified for all the synchronization objects in common, and unique attributes provided for each object.

Operating Synchronization Objects

Paired APIs are used for operating synchronization objects in principle; one is a wait API and the other is an API for releasing the wait. When a thread executes a wait API, if wait conditions are satisfied the thread does not lose the execution right and continues to be executed. Meanwhile, if the wait conditions are not satisfied the thread transitions to WAITING state and waits until the wait conditions are satisfied.

After a thread executes an API for releasing a wait, if there exists threads that satisfy the wait conditions the threads will wake up (transition to STANDBY or READY state) and become subject to scheduling. Owing to the scheduling, the thread that has executed the API for releasing a wait may be preempted and lost the execution right.

A time-out time can be specified for almost all wait APIs in a microsecond unit. A thread wakes up after a specified time has been elapsed regardless of the wait conditions. If a thread wakes up due to the timeout the thread will receive `SCE_KERNEL_ERROR_WAIT_TIMEOUT` from the wait API.

Deleting Synchronization Objects

Synchronization objects are deleted by using the delete API (`sceKernelDelete***()`) provided for each object. After the execution of the delete API, an identifier becomes invalid and cannot be used. Objects are usually deleted from the system at the time of execution of the delete API. However, when processing related to the object is being executed on another CPU, or when the object is being referenced (as described later), the object is not deleted until all references are completed.

When a synchronization object is deleted, a thread that has been waiting for the object wakes up and receives `SCE_KERNEL_ERROR_WAIT_DELETE`.

The identifier that can be passed as an argument to the delete API for synchronization objects is limited to the identifier obtained from the create API. The identifier obtained in the subsequently explained reference API cannot be passed as an argument. If it is wrongly passed, the `SCE_KERNEL_ERROR_UNKNOWN_***_ID` error will return.

Whether an arbitrary identifier was obtained from the creation or reference of a synchronization object can be evaluated by obtaining information of the identifier (`sceKernelGet***Info()`) and checking if the `SCE_UID_ATTR_OPEN_FLAG` bit is raised for an attribute (*`attr`* member of the obtained `SceKernel***Info` structure).

This evaluation can also be made by checking the same *`attr`* when using the debugger.

Cancelling Synchronization Objects

Some synchronization objects can be cancelled. When a synchronization object is cancelled, a thread that has been waiting for the object wakes up and receives `SCE_KERNEL_ERROR_WAIT_CANCEL`.

Referencing Synchronization Objects and Terminating the Reference

The reference API (`sceKernelOpen***()`) and the terminate reference API (`sceKernelClose***()`) are provided for some synchronization objects. If communication between processes is not performed it is not necessary to use these APIs since they are solely used for that purpose.

In order to reference synchronization objects, the `SCE_KERNEL_ATTR_OPENABLE` attribute should be specified when creating the objects. Note that it is necessary to define a unique name for an object within the system if specifying the attribute.

When a synchronization object is referenced, the object's new identifier (UID) is obtained and then operations can be performed by using the identifier. When terminating a reference, specify an identifier obtained at the time of reference and execute the terminate reference API.

Even when a delete API is executed while synchronization objects are being referenced, the synchronization objects will remain in the system until all references are completed. Moreover, the referencing side will not be affected by the delete API and will be able to continue using that synchronization object.

The identifier that can be passed as an argument to the terminate reference API for synchronization objects is limited to the identifier obtained from the reference API. The identifier obtained by the create API cannot be passed as an argument. If it is wrongly passed, the `SCE_KERNEL_ERROR_UNKNOWN_***_ID` error will return.

Whether an arbitrary identifier was obtained from the creation or reference of a synchronization object can be evaluated by obtaining information of the identifier (`sceKernelGet***Info()`) and checking if the `SCE_UID_ATTR_OPEN_FLAG` bit is raised for an attribute (*`attr`* member of the obtained `SceKernel***Info` structure).

This evaluation can also be made by checking the same *`attr`* when using the debugger.

Attributes of Synchronization Objects

Synchronization objects have attributes (*`attr`* argument and the *`attr`* member values of the `SceKernel***Info` structure) shared among all or some objects at the time the object is created.

Object Queue Rules

`SCE_KERNEL_ATTR_TH_FIFO` - FIFO queue
`SCE_KERNEL_ATTR_TH_PRIO` - Queue is in order of priority of waiting threads.

These are the object queue rules. For `SCE_KERNEL_ATTR_TH_FIFO`, the thread waiting for an object is awoken first. For `SCE_KERNEL_ATTR_TH_PRIO`, the highest priority thread waiting for an object is awoken. Multiple threads with the same priority run in FIFO order.

Type of Synchronization Objects

Characteristics and usage methods for each synchronization object are explained below.

Synchronization by Threads

Threads are also one of the synchronization objects. With `sceKernelWaitThreadEnd()`, a thread can wait for another thread to be terminated (enter DORMANT or DELETED state).

Thread Delay

In addition, the calling thread can be placed in WAITING state for a set time period with `sceKernelDelayThread()`. After the set time has been elapsed, the thread returns to READY state and is scheduled.

It is not necessary for an application to explicitly perform operations such as creation and deletion since the synchronization objects for thread delay are processed inside the kernel.

Event Flags

These are useful in programs that need to wake up many threads all at the same time. An event flag internally manages a flag with 32-bit pattern and recognizes a raised bit as indicating that an event is notified. Therefore, a single event flag can handle 32 different events separately. To use an event flag, first create one and set its initial pattern. Next, the thread that is to wait for some condition to be satisfied executes `sceKernelWaitEventFlag()` and specifies a cancellation pattern. This causes the thread to transition to WAITING state. When another thread or interrupt handler of the system executes `sceKernelSetEventFlag()` and updates the flag, the cancellation conditions of threads waiting on that event flag are compared against the updated value of the flag. If they match, the corresponding threads will be executable and scheduled.

Semaphores

When multiple threads are using the shared resources such as same device or data in the memory, the semaphore feature allows other threads to be notified of the usage status in order to provide synchronization. The concept of semaphore is like balls (semaphore resources) in a basket (semaphore object). To understand the concept, it would be helpful to imagine a ball being taken out of and returned to a basket. Threads that want to use shared resources, must acquire the semaphore resource by executing `sceKernelWaitSema()`. If the number of current semaphore resources is enough to be acquired the thread obtains the semaphore resources and uses the shared resources. Once the thread is done using the semaphore resource, `sceKernelSignalSema()` is executed and the semaphore resource is returned to the system.

If another thread is using the shared resources when `sceKernelWaitSema()` is executed, the semaphore resource will not be available, causing the calling thread to go into WAITING state and enter a semaphore waiting queue. When the thread using the shared resources executes `sceKernelSignalSema()` and returns the semaphore resource to the system after it is done, the returned semaphore resource will be given to the thread at the head of the semaphore waiting queue. When the number of semaphore resources reaches the number required by the thread, the thread returns to READY state and will be executed according to priority-based scheduling, thus allowing use of the shared resources.

The number of semaphore resources can be specified when a semaphore is created. Thus, in addition to providing exclusive control as described above, semaphores can also be used to indicate the valid data count in data buffers shared by multiple threads. When a thread supplies data that is stored in a shared buffer, `sceKernelSignalSema()` is executed to increase the valid data count. When a thread obtaining the data tries to fetch data from the shared buffer, `sceKernelWaitSema()` is executed. If the semaphore resource can be acquired, the data in the shared buffer is processed. If the resource cannot be acquired, the thread is put in WAITING state and waits for data to be provided.

Semaphore resources do not have the concept of ownership (owned thread), allowing a thread to execute `sceKernelSignalSema()` even if the thread has not acquired the semaphore resources.

Mutexes

A mutex, like a semaphore, allows other threads to be notified of the usage status of shared resources in order to provide synchronization, when multiple threads are using the shared resources such as the same device or data in the memory.

Threads that want to use a shared resource must acquire the mutex resource by executing `sceKernelLockMutex()`. Once the thread is done using the resource, `sceKernelUnlockMutex()` is executed and the mutex resource is returned to the system.

A mutex differs from a semaphore in that a mutex has the concept of ownership (owned thread) and while a semaphore counts the number of resources, a mutex is associated with only one resource and the mutex counts the number of times that the resource is locked. With a mutex, the resource is owned by the locked thread. Unlocking can only be performed by the thread that locked the mutex resource. Moreover when a mutex is locked recursively, the resource is not returned until the resource is unlocked the same number of times.

A mutex has the priority ceiling feature. By using this feature, while a mutex is being locked, the priority of the thread that locks the mutex is raised to a priority specified when the mutex is created. It is possible to reduce the possibility that a thread is preempted while locking a mutex, causing the mutex to remain locked for a long time.

If a thread terminates when a mutex resource it owns is still locked, the mutex is automatically unlocked.

Condition Variables

Condition variables are the feature that makes a thread wait until another thread reports that a certain state has been reached or a certain event has been generated.

Condition variables are always used in association with mutexes. A thread that is to wait on a condition variable must be the owner of the mutex associated with that condition variable. This mutex is unlocked automatically inside the kernel before the thread enters WAITING state by executing `sceKernelWaitCond()`. A thread awoken from WAITING state by another thread by executing `sceKernelSignalCond()` locks the mutex again upon awoken.

A thread can send a signal regardless of whether or not the thread has locked the mutex. However, if a signal is sent when there is no thread waiting for a condition variable, sending of the signal is valid, but it does not have any effect.

When multiple threads are waiting on a single condition variable and a signal is sent to that condition variable, the protocol attributes of the mutex associated with that condition variable are used to determine which thread is awoken first. For example, when the mutex protocol is FIFO, the thread waiting the longest on the condition variable is awoken and an attempt is made to obtain the mutex. However, when `sceKernelSignalCondAll()` is used to send a signal to all threads at the same time, the kernel scheduler determines which thread resumes processing first.

`sceKernelSignalCondTo()` can be used to send a condition signal by specifying a specific thread waiting on a condition variable with UID.

Reader/Writer Locks

Reader/writer locks are a synchronization object that provides a selectable mutual exclusiveness.

If threads that perform only reading operation (reader threads) use a shared resource simultaneously, there is no contention even during simultaneous parallel operation. Therefore, there is no need to protect by mutual exclusiveness. Mutual exclusiveness is required only when some of the threads execute the writing of data to a shared resource (writer threads).

By differentiating writer threads and reader threads, reader/writer locks aim to avoid this type of unnecessary mutual exclusiveness.

When a writer thread is not using a shared resource, a reader thread can use the shared resource at any time. On the other hand, only one writer thread can use a shared resource at any one time. When a writer thread is using a shared resource, all other threads, be they reader or writer threads, cannot use that shared resource.

The initialization sequence of the reader/writer locks and its usage is the same as that of mutexes. However, unlike a mutex, reader/writer locks have two types of locking (`sceKernelLockReadRWLock()` and `sceKernelLockWriteRWLock()`) and unlocking (`sceKernelUnlockReadRWLock()` and `sceKernelUnlockWriteRWLock()`) operations, one for the reader thread and one for the writer thread. A writer thread has the concept of ownership (owned thread), thus only the thread that owns the write lock can release the write lock. In addition, write locks support the recursive lock. Meanwhile, a read lock does not have the concept of ownership, enabling a thread to perform read unlock even if the thread does not own the read lock. However, it is not recommended to use this read lock method.

With regard to the wait operation of reader/writer locks, the write lock request is prioritized. That is, when a thread attempts to perform read lock with reader/writer locks, if there already exists a waiting thread to perform a write lock, the read lock request is placed after the write lock request in the waiting queue.

Lightweight Mutexes

A lightweight mutex is like an ordinary mutex except with faster lock and unlock times, because part of its work area is located in user memory.

Threads that want to use a shared resource must acquire the lightweight mutex resource by executing `sceKernelLockLwMutex()`. Once the thread is done using the resource, `sceKernelUnlockLwMutex()` is executed and the lightweight mutex resource is returned to the system.

When there is no contention to use a shared resource during a lock operation (the lock can be obtained immediately), or when a waiting thread does not exist upon unlocking (other threads need not to be woken up), processing is performed only in the user mode, enabling the processing to be faster.

Unlike a mutex, the work area for a lightweight mutex must be allocated in advance by the user. A wait cancellation operation cannot be performed for a thread that is waiting for a lightweight mutex by specifying that lightweight mutex. A lightweight mutex that is still locked when a thread terminates is not automatically unlocked. Furthermore, the priority ceiling feature is not provided for a lightweight mutex.

Lightweight Condition Variables

A lightweight condition variable is a condition variable used at the same time as a lightweight mutex.

Operations and capabilities are the same as above described (non-lightweight) condition variables. However, when a lightweight mutex is performed fast, the lightweight condition variable also becomes faster since both of them are used simultaneously.

Unlike a condition variable, the work area for a lightweight condition variable must be allocated in advance by the user, and a wait cancellation operation cannot be performed for a thread that is waiting for a lightweight condition variable by specifying that lightweight condition variable.

Event Objects

An event object is not a single synchronization object but is embedded in several kernel objects and provides shared synchronization features to those objects.

***Event objects are different from event flags.**

Synchronization objects with embedded event features are a message pipe, timer, etc. Kernel and driver modules other than the thread manager may also individually provide objects with embedded event features.

Event objects have a 32-bit event notification state. Each bit represents a specific "event" such as "data input/output", "timer firing" etc., and when a bit is raised, this indicates that an event is reported to an event object. Moreover, events in the `SCE_KERNEL_EVENT_USER_DEFINED_MASK` range are open for user-defined purposes, and users can arbitrarily perform notification by executing an event notification API such as `sceKernelSetEvent()` for various event objects.

The user can wait for a desired common event notification to objects with an embedded event feature by calling `sceKernelWaitEvent()`.

When both the pattern of the bit specified as the waiting event at this time and the pattern of the event reported to the event object are "0" and are no longer valid, the waiting thread is awoken.

This can be used in cases in which, for example, after the sending process to the message pipe is called in asynchronous mode, completion of the sending process is awaited.

In addition, the user can wait for event notification to different event objects at the same time by calling `sceKernelWaitMultipleEvents()`. AND wait/OR wait can be selected for multiple events with `sceKernelWaitMultipleEvents()`.

AND wait : Wait until all specified waiting events are completed or result in an error.
OR wait : Wait until one specified waiting event is completed or results in an error.

There are two types of event notifications: set notification and pulse notification.

The type of object determines which event notification is used. In some cases, it may be possible to specify the type of notification.

Set notification : The event is kept in the event object even after event notification.
Pulse notification : The event is not kept in the event object after event notification.

An event object has two attributes. Normally, the user specifies these when creating the object.

`SCE_KERNEL_EVENT_ATTR_MANUAL_RESET` : The event notification state is kept even when waiting of the thread corresponding to the reported event is completed.

`SCE_KERNEL_EVENT_ATTR_AUTO_RESET` : The event is cleared to non-notification state when individual waiting of the thread corresponding to the reported event is completed.

The following shows how event set notification, pulse notification and completion of event wait on a thread affect the state of event notifications.

`SCE_KERNEL_EVENT_ATTR_MANUAL_RESET`

Set notification : All threads waiting on a reported event are awoken.
Reported events are kept in notification state.
Pulse notification : All threads waiting on a reported event are awoken.
All reported events are cleared after the threads are awoken.
Wait completion : Event notification states are kept even after wait completion.

SCE_KERNEL_EVENT_ATTR_AUTO_RESET

Set notification : From the standpoint of the start of the waiting queue, waiting threads are awoken individually for each event.
Events that awaken threads after waiting is completed are not kept.

When multiple events complete waiting for a single waiting thread at the same time, the completed bits are all cleared at that time. *1

The reported events that did not awaken threads are kept until waiting is completed later on or until they are overwritten by other notification or by being cleared.

Pulse notification : From the standpoint of the start of the waiting queue, waiting threads are awoken individually for each notified event.

When multiple events complete waiting for a single waiting thread at the same time, the completed bits are all cleared at that time. *1

All reported events are cleared after the threads are awoken.

Wait completion : Only event notifications for events that have completed waiting are cleared.

*1 Note

Two threads (A and B) wait on the event object of a certain SCE_KERNEL_EVENT_ATTR_AUTO_RESET attribute, and the waiting events for each become as follows:

A:(0x00000001)

B:(0x80000001)

When two events are reported at the same time by set notification (0x80000001), if thread A is at the start of the waiting queue, both threads A and B wake up.

However, if thread B is at the start of the waiting queue, both events are cleared when thread B is awoken, and therefore, thread A does not wake up.

Event notification may involve the sending of 64-bit data (user data) defined for each object. The sent user data can be received by the waiting thread once waiting of the event object is completed.

Event objects keep the data last reported by an event.

For this reason, when multiple events are reported with no threads waiting, the user data of events reported first are overwritten by events reported later on.

The following events are defined by the kernel

The notification conditions of each event, as well as the value stored in user data upon notification, are explained below.

- SCE_KERNEL_EVENT_CREATE
  This is an event object create event.
  Each event object will be created in a state with this event notified.
  The identifier of the thread that created the applicable event object will be stored in the upper 32 bits of the user data, and the identifier of the process that created the applicable event object will be stored in the lower 32 bits of the user data.
- SCE_KERNEL_EVENT_DELETE
  This is an event object delete event.
  This event is notified to the deleted event object when an event object delete API is executed.
  The identifier of the thread that executed the delete API will be stored in the upper 32 bits of the user data, and the identifier of the process that executed the delete API will be stored in the lower 32 bits of the user data.

- SCE_KERNEL_EVENT_OPEN
  This is an event object reference event.
  This event is notified to the referenced event object when an event object reference API is executed.
  The identifier of the thread that executed the reference API will be stored in the upper 32 bits of the user data, and the identifier of the process that executed the reference API will be stored in the lower 32 bits of the user data.
- SCE_KERNEL_EVENT_CLOSE
  This is an event object terminate reference event.
  This event is notified to the reference-terminated event object when the event object terminate reference API is executed.
  The identifier of the thread that executed the terminate reference API will be stored in the upper 32 bits of the user data, and the identifier of the process that executed the terminate reference API will be stored in the lower 32 bits of the user data.
- SCE_KERNEL_EVENT_ERROR
  This event notifies that some kind of error occurred in an event object operation.
  The error code will be stored in the lower 32 bits of user data. The upper 32 bits will be defined per specific error or will be undefined.
- SCE_KERNEL_EVENT_TIMER
  This event notifies that the time specified to a timer has passed (timer trigger).
  For details, refer to the "Event Objects" item's "Timers" and the "Kernel Reference" document.
- SCE_KERNEL_EVENT_IN
  This event notifies that data of 1 byte or more has been placed in the message pipe's internal buffer.
  For details, refer to the "Event Objects" item's "Data exchange through message pipe" and the "Kernel Reference" document.
- SCE_KERNEL_EVENT_OUT
  This event notifies that data of 1 byte or more has been placed out of the message pipe's internal buffer.
  For details, refer to the "Event Objects" item's "Data exchange through message pipe" and the "Kernel Reference" document.

Simple Events

Simple events, which are the simplest event objects, have only an event notification/wait feature, and provide a feature similar to an event flag.

A thread that waits for event notification specifies an event in the SCE_KERNEL_EVENT_USER_DEFINED_MASK range and executes sceKernelWaitEvent() or sceKernelWaitMultipleEvents() to enter the WAITING state.

When another thread notifies an event using an event notification API such as sceKernelSetEvent() or sceKernelPulseEvent(), the threads that meet the wait conditions among the threads that have been waiting for the notified event become executable and are scheduled.

For details on the wait conditions and the event status following wait completion, refer to the "Event Objects" section.

The main difference with event flags is that multiple waits through combination with other event objects (sceKernelWaitMultipleEvents()) can be executed, and that the range of events that an application can arbitrarily notify/wait is limited to SCE_KERNEL_EVENT_USER_DEFINED_MASK.

The message pipes and timers described below include all the features of simple events.

Data exchange through message pipe

A message pipe enables data to be exchanged between processes and between threads. To exchange data, a message pipe copies data from a sending buffer to a receiving buffer. A message pipe is implemented as an event object.

To use a message pipe, first call `sceKernelCreateMsgPipe()` to create one. At this time, the internal buffer for the message pipe is allocated from the user memory budget. If a thread wishes to send a message, it calls `sceKernelSendMsgPipe()`. If the message can fit in the message pipe's internal buffer, the function returns immediately. If the message is too large, the sending thread enters WAITING state. The thread remains in WAITING state until another thread with a suitable receive buffer receives the message.

If a thread calls `sceKernelReceiveMsgPipe()`, to receive a message in the message pipe's internal buffer and the message size is such that it fills the receive buffer provided by the thread, or if there is a sending thread in WAITING state, the message will be obtained and the function will return immediately. If there is no pending message that exactly fills the receive buffer, the receiving thread will enter WAITING state until the sending thread can send the message.

Message data exchanged using a message pipe is not delimited. If a sending thread uses `sceKernelSendMsgPipe()` to send 10 bytes of data 10 times, a receiving thread can call `sceKernelReceiveMsgPipe()` to receive the entire 100 bytes of data all at once.

A message pipe can handle the non-continuous memory area as a send/receive buffer. In this case, an API with Vector such as `sceKernelSendMsgPipeVector()` is used.

In addition, PEEK operation can be performed for receiving the message pipe by specifying `SCE_KERNEL_MSG_PIPE_MODE_DONT_REMOVE` for the receiving mode. (Data remains in the internal buffer even after the data has been received.)

Sending and receiving the message pipe can also be done asynchronously. To send and receive the message pipe asynchronously, set `SCE_KERNEL_MSG_PIPE_MODE_DONT_WAIT` to the waiting mode for sending and receiving.

`sceKernelSendMsgPipe()` and `sceKernelReceiveMsgPipe()` return the process without waiting for sending and receiving to be completed.

Notification of an event wait (`sceKernelWaitEvent()`/`sceKernelWaitMultipleEvents()`) or a callback which is registered to a message pipe is used to wait for asynchronous sending and receiving to be completed. The following are the events notified as the operation result.

Sending complete	:	`SCE_KERNEL_EVENT_IN`
Receiving complete	:	`SCE_KERNEL_EVENT_OUT`
Sending/receiving error	:	`SCE_KERNEL_EVENT_ERROR`

The lower 32 bits of the user data at the completion of waiting represent the error code of the sending/receiving result, and the upper 32 bits represent the size of the data actually sent and received.

Event notification of the completion of sending/receiving for the message pipe is performed regardless of the waiting mode (even when not calling `SCE_KERNEL_MSG_PIPE_MODE_DONT_WAIT`).

Timers

Timers are provided as timers that the user can freely start, stop or change their count values. Since timers are independent of each another, an operation on one timer does not affect any other. The timer resolution is microsecond the same as that of the process time.

A timer is implemented as an event object, providing periodic callback notification and waking of threads. A timer is created with `sceKernelCreateTimer()` and started with `sceKernelStartTimer()` (count starts). When executing `sceKernelSetTimerEvent()` to set a timer event, the timer will issue the `SCE_KERNEL_EVENT_TIMER` event and carry out callback notification after a set time. User data attached to this event is 0.

Enabling the repeat setting creates a periodic timer that notifies events repeatedly.

The user can select whether to use set notification or pulse notification for the timer event notification.

Callback Feature

A callback is a means of conveying an asynchronous phenomenon to a thread. A thread generates a callback by setting up a callback function that will be called when the phenomenon occurs to `sceKernelCreateCallback()`. When the phenomenon occurs and is reported to the generated callback, the callback function that had previously been set up is executed. The callback function executes in the context of the thread that generated the callback.

An application reports the generation of a callback by executing `sceKernelNotifyCallback()`. In some cases, the system may make the report. When the generation of a callback is reported, its state is recorded in the thread that generated the callback.

The actual execution of the callback function is done at the time the thread that generated the callback checks for the existence of a callback notification by calling `sceKernelCheckCallback()`.

The thread that generated the callback is checked for the existence of a callback notification even if it execute a wait function with a CB suffix such as `sceKernelWaitSemaCB()`. In either case, the callback function is executed in the context of the thread that generated the callback.

If the callback function returns a value other than 0, that callback is automatically deleted. If 0 is returned, notification is received repeatedly.

Registering a Callback to an Event Object

A callback can be registered to an event object by calling `sceKernelRegisterCallbackToEvent()`. When an event object meets the callback notification conditions, notification is sent to the registered callback according to the `SCE_KERNEL_ATTR_NOTIFY_CB_xxx` attribute specified in the object. The callback notification conditions of the event objects depend on the individual event objects, but in most cases, they are the same as when reporting an event.

There is no limit to the number of callbacks that can be registered to an event object.

The callbacks registered to an event object are reported before the thread that is waiting on the event object is awoken. In this example, a thread has generated a callback and registered it to a message pipe, and the thread is waiting on that message pipe by calling `sceKernelReceiveMsgPipeCB()` (*with callback checking). When waking the thread and callback notification are performed at the same time by calling `sceKernelSendMsgPipe()`, the thread first executes the callback function and then `sceKernelReceiveMsgPipeCB()` is released.

Thread Events

The operations performed by the thread manager for starting and stopping threads are called thread events. Thread events are handled by specific handler functions, which are registered in advance. These handler functions are termed thread event handlers.

A thread event handler is registered by calling `sceKernelRegisterThreadEventHandler()`, and deleted by calling `sceKernelReleaseThreadEventHandler()`.

The two types of thread events shown below determine which thread event handler will be called. The thread event handler is executed in the thread context that generated the thread event.

> **Note**
> In this SDK release, thread event is not provided.

Thread startup event (TE_START)

Generated when `sceKernelStartThread()` is called to start a thread. The handler for this thread event is executed in the context of the started thread, immediately before it begins execution from its entry address.

Thread termination event (TE_EXIT)

Generated when either `sceKernelExitThread()` or `sceKernelExitDeleteThread()` is called to terminate a thread.

The handler for this thread event is executed in the context of the terminating thread.

Using VFP/NEON

Threads and interrupt handlers can use VFP/NEON by default.

Using TPIDRURW

TPIDRURW, a register unique to a thread, is initialized with 0 when the thread is created and is subject to context switching. Since the system does not use this register, it can be utilized for any purpose.

6 IO/File Manager

Overview

The PlayStation®Vita supports the following three device drivers.

(1) Resident library with its own API

(2) Resident module that can be used only from the IO/File manager's API

(3) Resident module that runs as a resident library with its own API, but can also be used via the IO/File manager's API

The IO/File manager enables an application program to call a device driver (the above type (2) or (3). The file access APIs include `sceIoOpen()`, `sceIoClose()`, `sceIoRead()`, and `sceIoWrite()`. The IO/File manager is also responsible for managing device driver registration.

Filenames Supported by the IO/File Manager

IO/File manager APIs such as `sceIoOpen()`, `sceIoDopen()`, `sceIoMkdir()`, or `sceIoRemove()` that require filenames or directory names give the following format to filenames.

device-name:filename-within-device

The filename-within-device, the string to the right of the colon (:), is a filename unique to each particular device driver. If a device does not use individual files internally, this string may not be present.

Filenames have the following restrictions.

- Paths are delimited by "/"
- Upper case and lower case characters are considered equivalent in a device name. For a filename, they may depend on the filesystem.

Examples of filenames

```
"tty0:"
"host0:/usr/local/sdk/module/xxx.sprx"
```

Notes on the Implementation of File Access Processing Taking into Consideration the File System Characteristics

File access by applications is to be made taking into consideration the characteristics of the file system, as explained below.

- For both access to files on the development host computer and access to the files on a memory card, it is recommended to use buffer alignment of 64 bytes specified with the *`buf`* argument when performing file access with the `sceIoRead()`/`sceIoPread()`/`sceIoWrite()`/`sceIoPwrite()` function. Buffer alignment of 64 bytes improves file access performance. Buffer alignment other than 64 bytes will result in a dramatic decline in performance. Moreover, if the buffer size exceeds 512 bytes, file access performance will improve in direct relation to the increase in buffer size.
- Do not place 100 or more files or directories in one directory. If the number of files and directories is likely to exceed 100, place them split among multiple directories.
- To perform random access, it is recommended to use the `sceIoPread()`/`sceIoPwrite()` function without calling the `sceIoLseek()` function. The system calls of `sceIoLseek()` can be eliminated by only calling `sceIoPread()`/`sceIoPwrite()` during random access.
- For both access to files on the development host computer and access to the files on a memory card, make the value specified for the *`offset`* argument a multiple of 512 when performing file access using the `sceIoPread()`/`sceIoPwrite()` function, as this improves the file access performance.