# Multi Thread Programming Tutorial



PlayStation®Vita Programmer Tool Runtime Library Release 3.50



Document serial number: 000004892117

Document serial number: 000004892117

# Table of Contents

# 1 Overview

For the execution of PlayStation®Vita applications, three CPU cores can be used. Therefore, it is important to execute processing of programs in parallel by actively using multi threads to enhance the performance of applications.

This tutorial explains the parallel programming environment available for PlayStation®Vita SDK and how to use the parallel programming

## Reference Materials

- Hardware Overview
- Programming Startup Guide
- Kernel Overview
- Kernel Reference
- libult Overview
- libult Reference
- samples/sample_code/system/tutorial_multi_threads/

# 2 Thread Management

## Types of Thread

For the PlayStation®Vita SDK, two types of threads, OS thread and user level thread, can be used.

The OS thread is managed by the OS. The user level thread is implemented on the OS thread and is managed at user level. Both types of threads can be used with a similar interface as for a general thread, but they differ from each other in terms of their performance and functionality.

As for the OS thread, the order of priority and the affinity for each thread can be specified, and the thread can synchronize with entities outside the process such as a device driver. On the other hand, the user level thread does not support several functions the OS thread supports such as the priority and the affinity, which is used for specifying the CPU core on which a thread is executed. In addition to its simple design, the user level thread does not need to call the system call for operating a thread other than the case when the parallelism equals to the number of the cores or lower, thus the cost for creation, switching or synchronization of a thread is low.

**Table 1 Comparison of OS Thread and User Level Thread**

| | OS Thread | User Level Thread |
|---|---|---|
| Thread management | Managed by OS | Managed at user level |
| Priority | Supports | Not support |
| Affinity | Supports | Not support |
| Synchronization with entities outside the process | Supports | Not support<br>(If synchronization is performed, a worker thread becomes subject to the thread management) |
| Usable synchronization object | Synchronization object of OS,<br>Synchronization object of libult | Synchronization object of libult<br>(If Synchronization object of OS is used, a worker thread becomes subject to the thread management) |
| Cost for creation or switching of a thread | High | Low |

## Processing Parallelization and Thread Selection

The following two points are important to enhance the performance.

- Increase the utilization ratio of the three cores
- Reduce the overhead of others than processing

The above two requirements are satisfied by dividing the whole processing into the number of cores and executing the processing on the same number of threads as the number of cores, if the threads do not stop during normal processing. But if the threads stop while devices are used or synchronization between threads is performed, or when the processing load cannot be divided equally, some thought are required for the processing.

### When processing includes synchronization with entities outside the process

When processing includes synchronization with entities outside the process (e.g. an operation for waiting the completion of IO device processing), the core utilization rate can be increased by preparing executable OS threads while waiting for the completion of the device processing, and using the core which is waiting for the device processing completion, because the thread stops and the core is released during the waiting time.

If such processing is done using the user level thread, the worker thread (an OS thread) stops while waiting for the completion of the device operation. The worker thread being in stop state does not execute any user level threads even if executable user level threads exist. Therefore, in the case like this, the core utilization rate is not increased even if there exist a number of executable user level threads.

Use the OS thread for the processing which includes synchronization with the external entities.

### When processing mainly handles calculations

When processing does not include exchanges with the outside, such as an IO, the core utilization rate can be increased by equally dividing the processing for cores. In the case of a loop processing, if the load for each loop remains constant, and the number of the loop is defined in advance, the load can be allocated equally by dividing the number of loop by the number of the core. On the other hand, in the case of a tree searching, which is not a simple loop processing, or in the case that the load for each loop is dynamically changed, equally dividing the processing before the execution is difficult. In these situations, processing load distribution using dynamic load balancing, which dynamically allocates the task to a core in which the processing has completed first, is required. By making the processing granularity small and increasing the number of threads, it is possible to perform the dynamic load balancing which allocates the threads to the core. The overhead of the dynamic load balancing can be reduced by using user level threads.

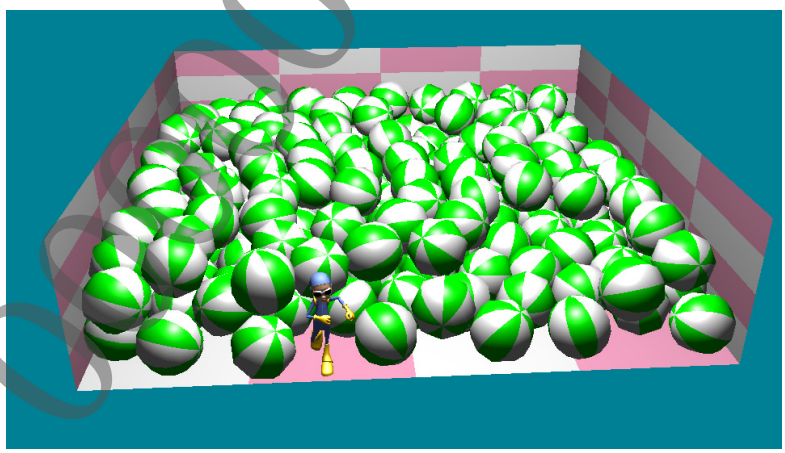
Processing parallelization for collision detection is explained below as an example of the dynamic load balancing. The programs are stored in "samples/sample_code/system/tutorial_multi_threads". The thread allocation method explained here can be tested by switching "Thread Configuration" in the menu of the sample.

This program executes collision detection of a number of spheres.

The algorism for collision detection is as described below.

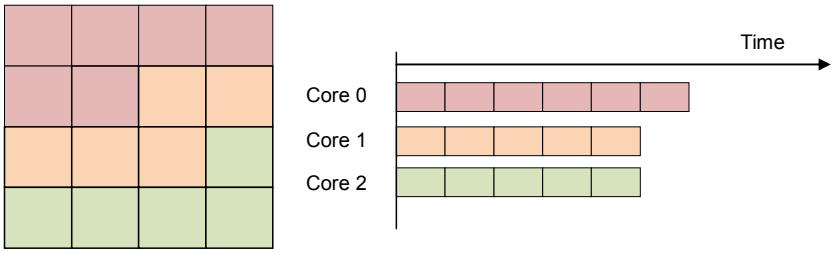
(1) Divide the whole space into 4x4 grids along the XZ plane
(2) Register each sphere to a grid corresponding to the XZ coordinates of the sphere
(3) Detect a collision in the grid and between the grid and adjoining grid for each grid
(4) Reflect the result of the collision detection to the sphere

The processing of (3) is the heaviest and can be executed in parallel. Therefore, this processing should be parallelized to improve the performance.



In the above example, it takes 37 msec to detect all collisions by using only one thread. By dividing the 16 grids into three and performing the collision detection on the three threads, Cores that have not been used for the one thread processing can be used. As a result, the core utilization rate is increased, and consequently the performance is improved.

**Figure 1 Division of Grids**




This method, however, has a disadvantage. In the situation as shown in the following example, the calculation time becomes longer because the load for the grids is uneven and then the load for the cores becomes unequal.

**Figure 2 Example of Uneven Processing (1)**

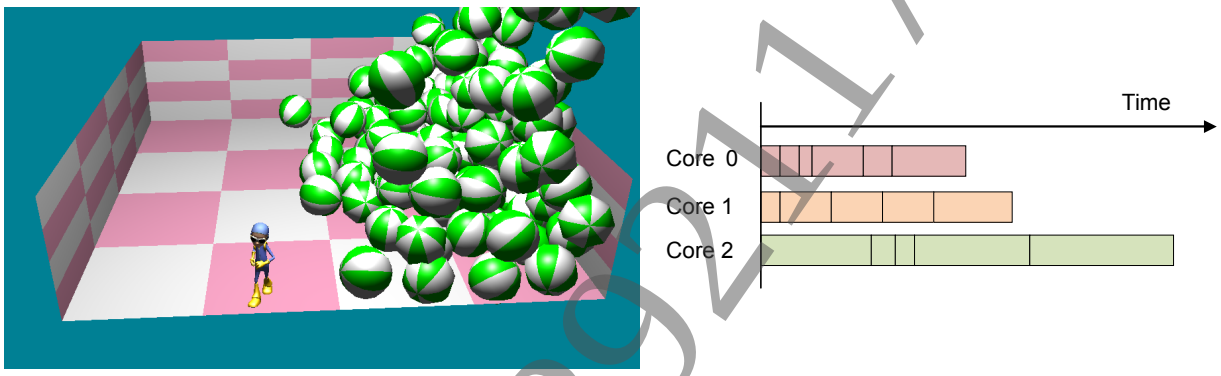
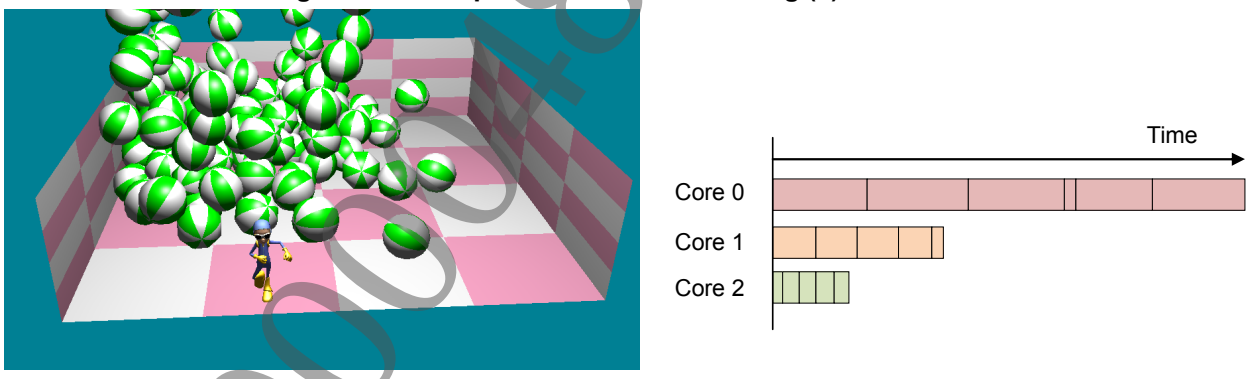
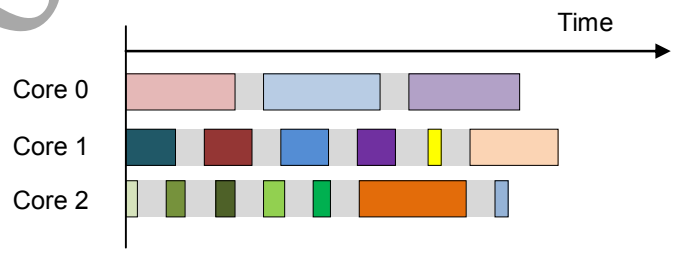



**Figure 3 Example of Uneven Processing (2)**




To address these situations, dynamic load balancing is required. A simple solution is to perform the load balancing by creating threads for each grid and then using the dynamic core allocation of the threads.

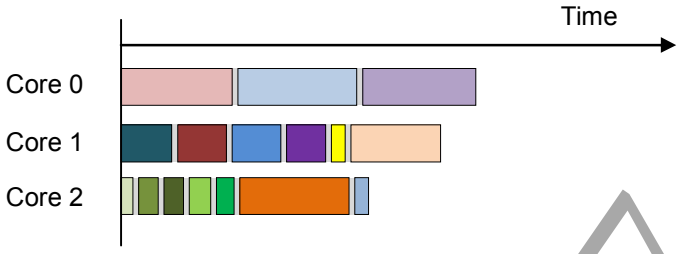
**Figure 4 Dynamic Load Balancing by Allocating Treads for Each Grid**



But, then the additional cost for thread switching affects the core utilization rate.

In this case, by using the user level threads instead of the OS threads, the overhead cased by the thread switching is reduced and the core utilization rate can be increased.

**Figure 5 When Using User Level Thread**




In the sample, when allocating the user level threads for each grid, the performance is approximately three times higher than the performance executed on one thread. The result shows the overhead caused by the thread switching is reduced to a certain level that does not affect the performance.

### Summary

When a synchronization with external entities is included (e.g. when using devices), there is no advantage to use user level threads because the synchronization is performed on a layer of OS thread. In addition, the user level threads do not have functions to fix the order of priority or the core to be executed. Therefore, the OS threads need to be used instead of user level threads when these functions are required.

On the other hand, when focusing on the management cost of the thread, such as dynamic load balancing, the overhead of the entire processing can be reduced by using the user level threads.

# 3 Synchronization

## Synchronization Library in the SDK

In the libraries provided by PlayStation®Vita SDK, the following are available for synchronization between threads.

- Synchronization provided by the OS
- Synchronization provided by libult
- Atomic operation

### Synchronization provided by the OS

This is a synchronization assumed to be used by OS threads. This synchronization has functions such as a function for retrieving mutexes in order of priority and can handle the priority of the OS threads correctly. If this synchronization is used on the user level thread, the worker thread, which is an OS thread executing the user level thread, stops the processing of user level threads during the wait state.

### Synchronization provided by libult

This is a synchronization assumed to be used by both OS threads and user level threads. The cost is low because the design is simple and this synchronization is mainly implemented at the user level. Especially when the operation for a synchronization object is immediately completed, or when switching the user level threads, all operations are performed with non-blocking processing, and no system calls are called. If the order of priority needs not to be considered, the overhead of synchronization can be reduced by using the libult synchronization.

### Atomic operation

This is an operation for variables using CPU atomic operation instructions provided by libatomic. Variables of data between the threads can be shared because this operation maintains a consistency between the threads. The atomic operation is the synchronization method which has the lowest cost, but only simple operations such as incrementing the shared integers can be handled. Please use the synchronization provided by libult when a thread control is required (e.g. blocking).

**Note**
If the synchronization is used based on busy waiting, a core cannot be given to other threads during the wait state, and consequently the core utilization rate is decreased. Also, in an execution environment in which the order of priority is fixed, a low-priority thread is not scheduled while a high-priority thread occupies a core, and therefore deadlock may easily be caused. Even if yield function is executed, a low-priority thread is never executed.
Moreover, if busy waiting is used, the electric power is highly consumed to stop the threads when there is no processing to be executed. Therefore, it is not recommended to use busy waiting.

## How to Use Synchronization in the Sample

The order of priority needs to be specified when two kinds of processing that requires and does not require real-time operations are mixed (e.g. background processing). In this sample, such kind of processing is not used, so the execution order of threads does not change the behavior. Therefore, the synchronization library of libult is basically used.

For the processing which is simple and requires multithread safe, such as registering a rigid body being in a grid to the grid or registering collision information to a rigid body, libatomic is used to realize a low-cost and multithread safe implementation.