Easy SPURS Overview

Table of Contents

1 Library Overview

Purpose and Characteristics

(1) Simplified SPURS APIs

Easy SPURS is a C++ class library provided to make it easier for developers to write codes using libspurs. It simplifies the complicated initialization procedure carried out by the interfaces of libspurs. Easy SPURS is just a wrapper of libspurs; it does not provide any additional features to SPURS.

(2) Exclusively for C++

Easy SPURS is exclusively for C++. It uses features specific to C++, such as, function overloading and default argument values, in order to simplify the steps for using libspurs.

(3) No dynamic internal memory allocation

Easy SPURS does not internally carry out dynamic memory allocation. Memory management is left up to the user.

(4) Simultaneous use with libspurs APIs

Easy SPURS does not hide libspurs APIs. For the features that are not supported by Easy SPURS, directly use the interfaces provided by libspurs. Easy SPURS is designed to be compatible with libspurs APIs. For example, in a function that requires a pointer to `CellSpurs`, a pointer to class Spurs can be passed instead.

Files

The following files are required for using Easy SPURS.

VisualStudio

Refer to the solution file as reference, and set dependency relationships. Add samples/common/spurs/ppu to the additional include directory, and add –lspurs_lib to the additional dependency file, to set a dependency relationship to spurs_util_ppu.vcproj and from spurs_util_ppu.vcproj to job/notify.vcproj.

Table 1 List of Files related to VisualStudio

Filename	Description
samples/common/spurs/Easy_SPURS.sln	Solution
samples/common/spurs/ppu/spurs_util_ppu.vcproj	Project
samples/common/spurs/ppu/job/notify.vcproj	Project
samples/common/spurs/ppu/spurs_util.h	Header file

Make

Add the include path and library path to samples/common/spurs/ppu/, and add –lspurs_util and lspurs_stub to the link target library. If you are using a common format makefile, include samples/common/spurs/libspurs_util.mk before the footer to perform the above processing.

Table 2 List of Files related to Make

Filename	Description
samples/common/spurs/ppu/spurs_util.h	Header files
samples/common/spurs/ppu/libspurs_util.a	Library files
samples/common/spurs/libspurs_util.mk	Common-format makefile as the makefile

Sample Programs

Sample programs using Easy SPURS can be compiled as follows.

Table 3 List of Easy SPURS Samples

Directory Name	Execution Filename	Description
samples/common/spurs/sample1_hello_job	hello_job.self	Easy job sample
samples/common/spurs/sample2_hello_task	hello_task.self	Easy task sample

VisualStudio

Open samples/common/spurs/Easy_Spurs.sln with VisualStudio and build the solution to compile all samples.

Make

Execute make in each sample's directory.

2 Using the Library

Basic Procedure

The procedure for using Easy SPURS consists of the following steps.

(1) Allocate memory
(2) Initialize
(3) Wait for completion and free memory

The steps for displaying "hello, world" on a SPURS task are exemplified below. Note that error handling has been omitted in this example to simplify the procedural steps.

```
#include <spurs_util.h>
using namespace cell::Util::Spurs;

// Step 1. Initialize the SPURS instance
Spurs* spurs = (Spurs*)std::memalign(Spurs::ALIGN, sizeof(Spurs));
Spurs::initialize(spurs, "sample");

// Step 2. Initialize the SPURS task set
Taskset* taskset = (Taskset*)std::memalign(Taskset::ALIGN, sizeof(Taskset));
Taskset::create(taskset, "sample", spurs, 0);

// Step 3. Create the SPURS task
Task* task = (Task*)std::memalign(Task::ALIGN, sizeof(Task));
Task::create(task, taskset, _binary_task_hello_elf_start, 0);

//
// "hello, world" appears
//

// Wait for the completion of the SPURS task
task->join();
std::free(task);

// Wait for the completion of the SPURS task set
taskset->shutdown();
taskset->join();
std::free(taskset);

// Terminate the SPURS instance
spurs->finalize();
std::free(spurs);
```

(1) Allocate memory

Allocate memory for classes accessed by SPUs (see Table 4) and their derivative classes (e.g. class CommandListDispatcher) where they can be accessed by SPUs. Do not create instances of these classes as automatic variables because it is impossible for SPUs to access the stack memory of a PPU.

Memory areas to be allocated must be aligned to the specified boundary. Because memory cannot be properly aligned when used, do not use the default `new` operator.

Table 4 Classes Accessed by SPUs

Class Name	Alignment	Initialize Method	Description
Spurs	Spurs::ALIGN	`initialize()`	Derivative of `CellSpurs`
JobChain	JobChain::ALIGN	`create()`	Derivative of `CellSpursJobChain`
JobGuard	JobGuard::ALIGN	`initialize()`	Derivative of `CellSpursJobGuard`
Job::Descriptor	16		Job descriptor base class
Job::Command	8		Job command base class
EventFlag	EventFlag::ALIGN	`initialize()`	Derivative of `CellSpursEventFlag`
Taskset	Taskset::ALIGN	`create()`	Derivative of `CellSpursTaskset`
Task	Task::ALIGN	`create()`	SPURS task termination synchronization

(2) Initialize

Before using the instances of classes with initializing method in Table 4 and their derivative classes (e.g. class CommandListDispatcher), make sure to initialize them using the respective initialization functions. There is no need to use placement new.

(3) Wait for completion and free memory

Before freeing the memory for an instance of class Spurs, class Taskset, class Task, class JobChain, or class CommandListDispatcher, make sure to check that it has been terminated. Any of these instances may be accessed by SPUs until its termination is confirmed.

Direct Use of libspurs APIs

Pointers to instances of classes accessed by SPUs (see Table 4) and their derivative classes are directly usable in functions provided by libspurs. For example, a pointer to an instance of class Spurs can be used in a SPURS job chain.

```
#include <spurs_util.h>
using namespace cell::Util::Spurs;

Spurs* spurs = (Spurs*)std::memalign(Spurs::ALIGN, sizeof(Spurs));
Spurs::initialize(spurs, "sample")

CellSpursJobChain* jobChain = (CellSpursJobChain*)memalign(128,
sizeof(CellSpursJobChain));
CellSpursJobChainAttribute attribute;
/*
 * snip initializing attributes
 */
cellSpursCreateJobChainWithAttribute(spurs, jobChain, &attribute);
```

3 Reducing the Context Save Area of a SPURS Task

Background

A SPURS task requires memory for saving its context when waiting for the progress of some other processing using the synchronization library. Close to 240 KB is required to save its entire context. However, it is possible to reduce the total area to be saved by selectively saving certain parts of the context.

Overview

Class TaskElf uses ELF segment information of a SPURS task, which can also be referenced from the PPU, to enable the specification of the LS context areas to save. ELF segment information refers to the information that is generally used to load a program; note that it is different from the ELF section information. SPURS task modules also use this ELF segment information when transferring SPURS tasks from main memory onto the LS.

Operational Procedure

This section describes operational procedure using an example of a SPURS task with the following ELF segment layout.

```
% spu-lv2-readelf -l task/wait.elf

Elf file type is EXEC (Executable file)
Entry point 0x3088
There are 3 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg  Align
  LOAD           0x000100 0x00003000 0x00003000 0x00ac0 0x00ac0 R E  0x80
  LOAD           0x000c00 0x00003b00 0x00003b00 0x00010 0x00020 RW   0x80
  NOTE           0x000c10 0x00000000 0x00000000 0x00034 0x00034 R    0x4

 Section to Segment mapping:
  Segment Sections...
   00     .SpuGUID .text .rodata
   01     .ctors .dtors .bss
   02     .note.spu_name
```

ELF segments are listed under `Program Headers`. A segment whose `Type` is `LOAD` and `Flg` is `R`, is read-only. A segment whose `Type` is `LOAD` and `Flg` is `RW`, is a read-write enabled segment.

Exclude read-only segments

The simplest approach to reducing the amount of memory for saving a context is to exclude the read-only ELF segments. When executing a task whose context has been partially saved, the SPURS task module will automatically load the read-only segments from the ELF. Thus, this approach enables the save area to be reduced without applying any restrictions to the program of the SPURS task.

```
#include <spurs_util.h>
using namespace cell::Util::Spurs;
extern const char _binary_task_wait_elf_start[];

TaskElf elf(_binary_task_wait_elf_start);
elf.setAll();
elf.unsetReadOnlySegment();
```

```
unsigned size = elf.saveBufferSize();
void* buf = std::memalign(TaskElf::SaveBufferAlign(), size);
int ret = Task::create(task, taskset, elf, 0, buf, size));
assert(ret == CELL_OK);
// ...
ret = task->join();
assert(ret == CELL_OK);
std::free(buf);
```

Exclude read-only segments and the heap memory

Another approach is to save writable ELF segments and the stack memory. Read-only segments and the heap memory are not saved. This approach is effective on programs that do not use heap memory functions, such as, `malloc()` and the `new` operator.

```
#include <spurs_util.h>
using namespace cell::Util::Spurs;
extern const char _binary_task_wait_elf_start[];

TaskElf elf(_binary_task_wait_elf_start);
elf.unsetAll();
elf.setWritableSegment();
elf.setStack(2048);

unsigned size = elf.saveBufferSize();
void* buf = std::memalign(TaskElf::SaveBufferAlign(), size);
int ret = Task::create(task, taskset, elf, 0, buf, size));
assert(ret == CELL_OK);
// ...
ret = task->join();
assert(ret == CELL_OK);
std::free(buf);
```