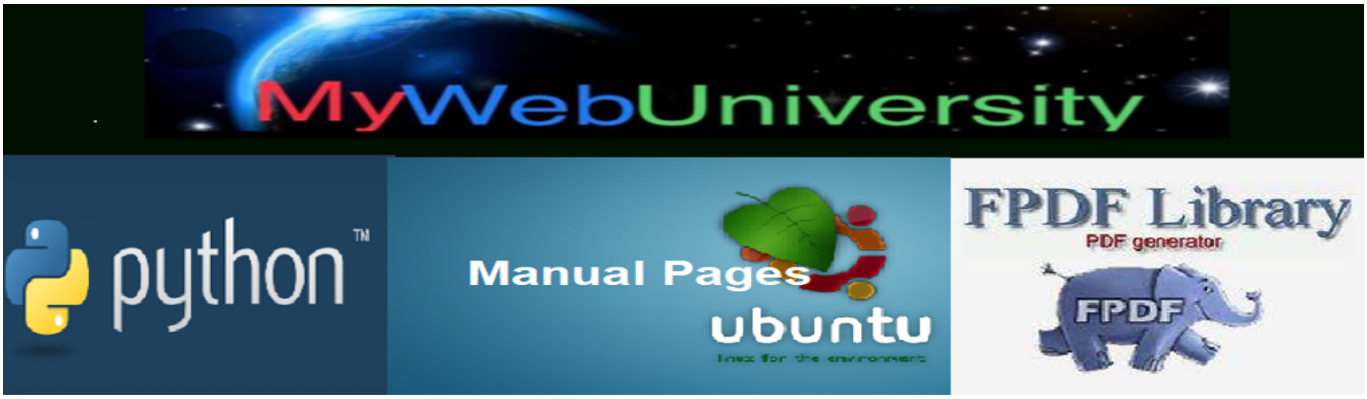

Rocky Enterprise Linux 9.2 Manual Pages on command 'spu_run.2'

$ man spu_run.2

SPU_RUN(2) Linux Programmer's Manual SPU_RUN(2)

NAME

spu_run - execute an SPU context

SYNOPSIS

#include <sys/spu.h>

int spu_run(int fd, unsigned int *npc, unsigned int *event);

Note: There is no glibc wrapper for this system call; see NOTES.

DESCRIPTION

The spu_run() system call is used on PowerPC machines that implement the Cell Broadband Engine Architecture in order to access Synergistic Processor Units (SPUs). The fd argu? ment is a file descriptor returned by spu_create(2) that refers to a specific SPU context.

When the context gets scheduled to a physical SPU, it starts execution at the instruction pointer passed in npc.

Execution of SPU code happens synchronously, meaning that spu_run() blocks while the SPU is still running. If there is a need to execute SPU code in parallel with other code on either the main CPU or other SPUs, a new thread of execution must be created first (e.g., using pthread_create(3)).

When spu_run() returns, the current value of the SPU program counter is written to npc, so successive calls to spu_run() can use the same npc pointer.

The event argument provides a buffer for an extended status code. If the SPU context was created with the SPU_CREATE_EVENTS_ENABLED flag, then this buffer is populated by the Linux kernel before spu_run() returns.

The status code may be one (or more) of the following constants:

SPE_EVENT_DMA_ALIGNMENT

A DMA alignment error occurred.

SPE_EVENT_INVALID_DMA

An invalid MFC DMA command was attempted.

SPE_EVENT_SPE_DATA_STORAGE

A DMA storage error occurred.

SPE_EVENT_SPE_ERROR

An illegal instruction was executed.

NULL is a valid value for the event argument. In this case, the events will not be re? ported to the calling process.

RETURN VALUE

On success, spu_run() returns the value of the spu_status register. On error, it returns -1 and sets errno to one of the error codes listed below.

The spu_status register value is a bit mask of status codes and optionally a 14-bit code returned from the stop-and-signal instruction on the SPU. The bit masks for the status codes are:

0x02 SPU was stopped by a stop-and-signal instruction.

0x04 SPU was stopped by a halt instruction.

0x08 SPU is waiting for a channel.

0x10 SPU is in single-step mode.

0x20 SPU has tried to execute an invalid instruction.

0x40 SPU has tried to access an invalid channel.

0x3fff0000

The bits masked with this value contain the code returned from a stop-and-signal instruction. These bits are valid only if the 0x02 bit is set.

If spu_run() has not returned an error, one or more bits among the lower eight ones are always set.

ERRORS

EBADF fd is not a valid file descriptor.

EFAULT npc is not a valid pointer, or event is non-NULL and an invalid pointer.

EINTR A signal occurred while spu_run() was in progress; see signal(7). The npc value has been updated to the new program counter value if necessary.

EINVAL fd is not a valid file descriptor returned from spu_create(2).

ENOMEM There was not enough memory available to handle a page fault resulting from a Mem?
ory Flow Controller (MFC) direct memory access.

ENOSYS The functionality is not provided by the current system, because either the hard?
ware does not provide SPUs or the spufs module is not loaded.

VERSIONS

The spu_run() system call was added to Linux in kernel 2.6.16.

CONFORMING TO

This call is Linux-specific and implemented only by the PowerPC architecture. Programs using this system call are not portable.

NOTES

Glibc does not provide a wrapper for this system call; call it using syscall(2). Note however, that spu_run() is meant to be used from libraries that implement a more abstract interface to SPUs, not to be used from regular applications. See ?http://www.bsc.es/projects/deepcomputing/linuxoncell/? for the recommended libraries.

EXAMPLES

The following is an example of running a simple, one-instruction SPU program with the spu_run() system call.

```
#include <stdlib.h>
#include <stdint.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)
int main(void)
{
    int context, fd, spu_status;
    uint32_t instruction, npc;
    context = spu_create("/spu/example-context", 0, 0755);
    if (context == -1)
        handle_error("spu_create");
    /* write a 'stop 0x1234' instruction to the SPU's
```

```
         * local store memory
         */
        instruction = 0x00001234;
        fd = open("/spu/example-context/mem", O_RDWR);
        if (fd == -1)
            handle_error("open");
        write(fd, &instruction, sizeof(instruction));
        /* set npc to the starting instruction address of the
         * SPU program. Since we wrote the instruction at the
         * start of the mem file, the entry point will be 0x0
         */
        npc = 0;
        spu_status = spu_run(context, &npc, NULL);
        if (spu_status == -1)
            handle_error("open");
        /* we should see a status code of 0x1234002:
         *   0x00000002 (spu was stopped due to stop-and-signal)
         * | 0x12340000 (the stop-and-signal code)
         */
        printf("SPU Status: %#08x\n", spu_status);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

close(2), spu_create(2), capabilities(7), spufs(7)

COLOPHON

This page is part of release 5.10 of the Linux man-pages project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.