NAME
> spufs - SPU filesystem

DESCRIPTION
> The SPU filesystem is used on PowerPC machines that implement the Cell Broadband Engine Architecture in order to access Synergistic Processor Units (SPUs).
>
> The filesystem provides a name space similar to POSIX shared memory or message queues. Users that have write permissions on the filesystem can use spu_create(2) to establish SPU contexts under the **spufs** root directory.
>
> Every SPU context is represented by a directory containing a predefined set of files. These files can be used for manipulating the state of the logical SPU. Users can change permissions on the files, but can't add or remove files.

> **Mount options**
> > **uid=<uid>**
> > > Set the user owning the mount point; the default is 0 (root).
> >
> > **gid=<gid>**
> > > Set the group owning the mount point; the default is 0 (root).
> >
> > **mode=<mode>**
> > > Set the mode of the top-level directory in **spufs**, as an octal mode string. The default is 0775.

> **Files**
> > The files in **spufs** mostly follow the standard behavior for regular system calls like read(2) or write(2), but often support only a subset of the operations supported on regular filesystems. This list details the supported operations and the deviations from the standard behavior described in the respective man pages.
> >
> > All files that support the read(2) operation also support readv(2) and all files that support the write(2) operation also support writev(2). All files support the access(2) and stat(2) family of operations, but for the latter call, the only fields of the returned *stat* structure that contain reliable information are *st_mode*, *st_nlink*, *st_uid*, and *st_gid*.
> >
> > All files support the chmod(2)/ fchmod(2) and chown(2)/ fchown(2) operations, but will not be able to grant permissions that contradict the possible operations (e.g., read access on the *wbox* file).
> >
> > The current set of files is:
> >
> > */capabilities*
> > > Contains a comma-delimited string representing the capabilities of this SPU context. Possible capabilities are:
> > >
> > > **sched**   This context may be scheduled.
> > >
> > > **step**   This context can be run in single-step mode, for debugging.
> > >
> > > New capabilities flags may be added in the future.
> >
> > */mem*   the contents of the local storage memory of the SPU. This can be accessed like a regular shared memory file and contains both code and data in the address space of the SPU. The possible operations on an open *mem* file are:
> > >
> > > **read**(2), **pread**(2), **write**(2), **pwrite**(2),
> > > > lseek(2) These operate as usual, with the exception that lseek(2), write(2), and pwrite(2) are not supported beyond the end of the file. The file size is the size of the local storage of the SPU, which is normally 256 kilobytes.
> > > >
> > > > mmap(2) Mapping *mem* into the process address space provides access to the SPU local storage within the process address space. Only **MAP_SHARED** mappings are allowed.
> >
> > */regs*   Contains the saved general-purpose registers of the SPU context. This file contains the 128-bit values of each register, from register 0 to register 127, in order. This allows the general-purpose registers to be inspected for debugging.

Reading to or writing from this file requires that the context is scheduled out, so use of this file is not recommended in normal program operation.

The *regs* file is not present on contexts that have been created with the **SPU_CRE-ATE_NOSCHED** flag.

/mbox The first SPU-to-CPU communication mailbox. This file is read-only and can be read in units of 4 bytes. The file can be used only in nonblocking mode - even poll(2) cannot be used to block on this file. The only possible operation on an open *mbox* file is:

> read(2) If *count* is smaller than four, read(2) returns -1 and sets *errno* to **EINVAL**. If there is no data available in the mailbox (i.e., the SPU has not sent a mailbox message), the return value is set to -1 and *errno* is set to **EAGAIN**. When data has been read successfully, four bytes are placed in the data buffer and the value four is returned.

/ibox The second SPU-to-CPU communication mailbox. This file is similar to the first mailbox file, but can be read in blocking I/O mode, thus calling read(2) on an open *ibox* file will block until the SPU has written data to its interrupt mailbox channel (unless the file has been opened with **O_NONBLOCK**, see below). Also, poll(2) and similar system calls can be used to monitor for the presence of mailbox data.

The possible operations on an open *ibox* file are:

> read(2) If *count* is smaller than four, read(2) returns -1 and sets *errno* to **EINVAL**. If there is no data available in the mailbox and the file descriptor has been opened with **O_NONBLOCK**, the return value is set to -1 and *errno* is set to **EAGAIN**.

> If there is no data available in the mailbox and the file descriptor has been opened without **O_NONBLOCK**, the call will block until the SPU writes to its interrupt mailbox channel. When data has been read successfully, four bytes are placed in the data buffer and the value four is returned.

> poll(2) Poll on the *ibox* file returns *(POLLIN | POLLRDNORM)* whenever data is available for reading.

/wbox The CPU-to-SPU communication mailbox. It is write-only and can be written in units of four bytes. If the mailbox is full, write(2) will block, and poll(2) can be used to block until the mailbox is available for writing again. The possible operations on an open *wbox* file are:

> write(2) If *count* is smaller than four, write(2) returns -1 and sets *errno* to **EINVAL**. If there is no space available in the mailbox and the file descriptor has been opened with **O_NONBLOCK**, the return value is set to -1 and *errno* is set to **EAGAIN**.

> If there is no space available in the mailbox and the file descriptor has been opened without **O_NONBLOCK**, the call will block until the SPU reads from its PPE (PowerPC Processing Element) mailbox channel. When data has been written successfully, the system call returns four as its function result.

> poll(2) A poll on the *wbox* file returns *(POLLOUT | POLLWRNORM)* whenever space is available for writing.

/mbox_stat, */ibox_stat*, */wbox_stat*
These are read-only files that contain the length of the current queue of each mailbox—that is, how many words can be read from *mbox* or *ibox* or how many words can be written to *wbox* without blocking. The files can be read only in four-byte units and return a big-endian binary integer number. The only possible operation on an open *box_stat* file is:

> read(2) If *count* is smaller than four, read(2) returns -1 and sets *errno* to **EINVAL**. Otherwise, a four-byte value is placed in the data buffer. This value is the number of elements that can be read from (for *mbox_stat* and *ibox_stat*) or written to (for *wbox_stat*) the respective mailbox without blocking or returning an **EAGAIN** error.

/npc, */decr*, */decr_status*, */spu_tag_mask*, */event_mask*, */event_status*, */srr0*, */lslr*

Internal registers of the SPU. These files contain an ASCII string representing the hex value of the specified register. Reads and writes on these files (except for *npc*, see below) require that the SPU context be scheduled out, so frequent access to these files is not recommended for normal program operation.

The contents of these files are:

npc   Next Program Counter - valid only when the SPU is in a stopped state.

decr   SPU Decrementer

decr_status   Decrementer Status

spu_tag_mask   MFC tag mask for SPU DMA

event_mask   Event mask for SPU interrupts

event_status   Number of SPU events pending (read-only)

srr0   Interrupt Return address register

lslr   Local Store Limit Register

The possible operations on these files are:

read(2) Reads the current register value. If the register value is larger than the buffer passed to the read(2) system call, subsequent reads will continue reading from the same buffer, until the end of the buffer is reached.

When a complete string has been read, all subsequent read operations will return zero bytes and a new file descriptor needs to be opened to read a new value.

write(2) A write(2) operation on the file sets the register to the value given in the string. The string is parsed from the beginning until the first nonnumeric character or the end of the buffer. Subsequent writes to the same file descriptor overwrite the previous setting.

Except for the *npc* file, these files are not present on contexts that have been created with the **SPU_CREATE_NOSCHED** flag.

/fpcr   This file provides access to the Floating Point Status and Control Register (fcpr) as a binary, four-byte file. The operations on the *fpcr* file are:

read(2) If *count* is smaller than four, read(2) returns -1 and sets *errno* to **EINVAL**. Otherwise, a four-byte value is placed in the data buffer; this is the current value of the *fpcr* register.

write(2) If *count* is smaller than four, write(2) returns -1 and sets *errno* to **EINVAL**. Otherwise, a four-byte value is copied from the data buffer, updating the value of the *fpcr* register.

/signal1, */signal2*

The files provide access to the two signal notification channels of an SPU. These are read-write files that operate on four-byte words. Writing to one of these files triggers an interrupt on the SPU. The value written to the signal files can be read from the SPU through a channel read or from host user space through the file. After the value has been read by the SPU, it is reset to zero. The possible operations on an open *signal1* or *signal2* file are:

read(2) If *count* is smaller than four, read(2) returns -1 and sets *errno* to **EINVAL**. Otherwise, a four-byte value is placed in the data buffer; this is the current value of the specified signal notification register.

write(2) If *count* is smaller than four, write(2) returns -1 and sets *errno* to **EINVAL**. Otherwise, a four-byte value is copied from the data buffer, updating the value of the specified signal notification register. The signal notification register will either be replaced with the input data or will be updated to the bitwise OR operation of the old

value and the input data, depending on the contents of the *signal1_type* or *signal2_type* files respectively.

/signal1_type, */signal2_type*

These two files change the behavior of the *signal1* and *signal2* notification files. They contain a numeric ASCII string which is read as either "1" or "0". In mode 0 (overwrite), the hardware replaces the contents of the signal channel with the data that is written to it. In mode 1 (logical OR), the hardware accumulates the bits that are subsequently written to it. The possible operations on an open *signal1_type* or *signal2_type* file are:

read(2) When the count supplied to the read(2) call is shorter than the required length for the digit (plus a newline character), subsequent reads from the same file descriptor will complete the string. When a complete string has been read, all subsequent read operations will return zero bytes and a new file descriptor needs to be opened to read the value again.

write(2) A write(2) operation on the file sets the register to the value given in the string. The string is parsed from the beginning until the first nonnumeric character or the end of the buffer. Subsequent writes to the same file descriptor overwrite the previous setting.

/mbox_info, */ibox_info*, */wbox_info*, */dma_into*, */proxydma_info*

Read-only files that contain the saved state of the SPU mailboxes and DMA queues. This allows the SPU status to be inspected, mainly for debugging. The *mbox_info* and *ibox_info* files each contain the four-byte mailbox message that has been written by the SPU. If no message has been written to these mailboxes, then contents of these files is undefined. The *mbox_stat*, *ibox_stat* and *wbox_stat* files contain the available message count.

The *wbox_info* file contains an array of four-byte mailbox messages, which have been sent to the SPU. With current CBEA machines, the array is four items in length, so up to 4 * 4 = 16 bytes can be read from this file. If any mailbox queue entry is empty, then the bytes read at the corresponding location are undefined.

The *dma_info* file contains the contents of the SPU MFC DMA queue, represented as the following structure:

```
struct spu_dma_info {
uint64_t dma_info_type;
uint64_t dma_info_mask;
uint64_t dma_info_status;
uint64_t dma_info_stall_and_notify;
uint64_t dma_info_atomic_command_status;
struct mfc_cq_sr dma_info_command_data[16];
};
```

The last member of this data structure is the actual DMA queue, containing 16 entries. The *mfc_cq_sr* structure is defined as:

```
struct mfc_cq_sr {
uint64_t mfc_cq_data0_RW;
uint64_t mfc_cq_data1_RW;
uint64_t mfc_cq_data2_RW;
uint64_t mfc_cq_data3_RW;
};
```

The *proxydma_info* file contains similar information, but describes the proxy DMA queue (i.e., DMAs initiated by entities outside the SPU) instead. The file is in the following format:

```
struct spu_proxydma_info {
uint64_t proxydma_info_type;
uint64_t proxydma_info_mask;
```

```
uint64_t proxydma_info_status;
struct mfc_cq_sr proxydma_info_command_data[8];
};
```

Accessing these files requires that the SPU context is scheduled out - frequent use can be inefficient. These files should not be used for normal program operation.

These files are not present on contexts that have been created with the **SPU_CRE-ATE_NOSCHED** flag.

/cntl   This file provides access to the SPU Run Control and SPU status registers, as an ASCII string. The following operations are supported:

> read(2) Reads from the *cntl* file will return an ASCII string with the hex value of the SPU Status register.

> write(2) Writes to the *cntl* file will set the context's SPU Run Control register.

/mfc   Provides access to the Memory Flow Controller of the SPU. Reading from the file returns the contents of the SPU's MFC Tag Status register, and writing to the file initiates a DMA from the MFC. The following operations are supported:

> write(2) Writes to this file need to be in the format of a MFC DMA command, defined as follows:

```
struct mfc_dma_command {
int32_t pad; /* reserved */
uint32_t lsa; /* local storage address */
uint64_t ea; /* effective address */
uint16_t size; /* transfer size */
uint16_t tag; /* command tag */
uint16_t class; /* class ID */
uint16_t cmd; /* command opcode */
};
```

> Writes are required to be exactly *sizeof(struct mfc_dma_command)* bytes in size. The command will be sent to the SPU's MFC proxy queue, and the tag stored in the kernel (see below).

> read(2) Reads the contents of the tag status register. If the file is opened in blocking mode (i.e., without **O_NONBLOCK**), then the read will block until a DMA tag (as performed by a previous write) is complete. In nonblocking mode, the MFC tag status register will be returned without waiting.

> poll(2) Calling poll(2) on the *mfc* file will block until a new DMA can be started (by checking for **POLLOUT**) or until a previously started DMA (by checking for **POLLIN**) has been completed.

> */mss* Provides access to the MFC MultiSource Synchronization (MSS) facility. By mmap(2)-ing this file, processes can access the MSS area of the SPU.

> The following operations are supported:

> mmap(2) Mapping **mss** into the process address space gives access to the SPU MSS area within the process address space. Only **MAP_SHARED** mappings are allowed.

/psmap   Provides access to the whole problem-state mapping of the SPU. Applications can use this area to interface to the SPU, rather than writing to individual register files in **spufs**.

The following operations are supported:

> mmap(2) Mapping **psmap** gives a process a direct map of the SPU problem state area. Only **MAP_SHARED** mappings are supported.

/phys-id
> Read-only file containing the physical SPU number that the SPU context is running on. When the context is not running, this file contains the string "-1".

> The physical SPU number is given by an ASCII hex string.

/object-id
> Allows applications to store (or retrieve) a single 64-bit ID into the context. This ID is later used by profiling tools to uniquely identify the context.

>> write(2) By writing an ASCII hex value into this file, applications can set the object ID of the SPU context. Any previous value of the object ID is overwritten.

>> read(2) Reading this file gives an ASCII hex string representing the object ID for this SPU context.

EXAMPLE
> */etc/fstab* entry
>> none /spu spufs gid=spu 0 0

SEE ALSO
> close(2), spu_create(2), spu_run(2), capabilities(7)

> *The Cell Broadband Engine Architecture (CBEA) specification*

COLOPHON
> This page is part of release 4.10 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/.