SessionFS
A session based VFS wrapper

Advanced Operating Systems and Virtualization
A.A. 2018/2019
Final Project

MATTIA NICOLELLA 1707844

SAPIENZA
UNIVERSITÀ DI ROMA

Chapter 1

Final Project Report

Student Full Name: Mattia Nicolella **Student ID:** 1707844

The following report will consist on a general introduction on the content of the project and several sections that will describe what each module implements. The detailed description of how the functions are implemented is provided in the file section of the documentation and it will be referenced during the report when these details are necessary.

The documentation, including this report, is generated by Doxygen and is provided in PDF and HTML format under the `docs` folder with the following filename: `Documentation`.

1.1 Introduction

SessionFS is a virtual file system wrapper which allows an userspace application to use the Unix session semantics to operate on files.

To enable the Unix session semantic, the `open` and `close` system calls are wrapped by a userspace library (that must be directly included or at least preloaded) which will check if the invocation is a "normal" one or if uses the Unix session semantic.

If the invocation of the system call is not a "normal" one and thus uses the `O_SESS` custom flag with the "correct" path, the userspace library will execute an ioctl to an ad-hoc virtual character device, called `SessionFS_dev`, which will handle the invocation accordingly.

If the `O_SESS` flag is missing, or the path is not the one in which sessions are allowed the userspace library will use the libc version of the called function.

This allows the userspace application to trigger the usage of the Unix session semantic with the custom defined `O_SESS` flag when opening a file in the correct path and does not change the semantic for closing a file.

The module allows the semantic to be used inside a folder which can be specified or changed by the userspace application anytime (defaults to `/mnt`) without having effects on the already opened sessions. To change the session path is only necessary to write on the `SessionFS_dev` device file the new absolute path in which sessions will be enabled or using the `write_sess_path()` function provided by the library. To get the current session path instead, a read on the `SessionFS_dev` device file is needed; as before, the userspace library provides the `get_sess_path()` to do so.

On kernel level, the type of ioctl received determines the operation to be executed by another module, the session manager. The session manger handles the creation of the new session and the closure of existing sessions. For each file which has active sessions it will enforce the Unix session semantic when adding and deleting incarnations.

Adding an incarnation requires copying the information from the original file to an incarnation file which will have the same filenname but will end with `_incarnation_[pid]_[timestamp]`. Deleting an incarnation requires that the content of the incarnation file will be copied over the original file, overwriting its content; naturally creating an incarnation while another is being closed is not possible, so these operations will be serialized. Is possible, instead, to open concurrently multiple incarnations from the same original file.

The last module involved is the session info module, which will publish information on the active sessions via SysFS. In detail we have a pseudofile under `/sys/devices/virtual/SessionFS_dev` called `active←_sessions` which contains the number of active incarnations. For each original file we have a folder in the same location as the previous pseudofile with the path of the original file where slashes are substituted by dashes (`/mnt/file` becomes `-mnt-file`); in this folder we have a pseudo file for each process that has an active incarnation on that original file. These pseudofiles have as the filename the pid of the process and when read they provide the name of the process with that pid.

Finally the session manager is also able to determine if there are invalid sessions, which are owned by dead processes and remove them.

1.2 User Space Library

This library (libsessionfs.c) wraps the `open` and `close` systemcalls to determine which semantic is used during invocation if the`SessionFS_dev` device must bu used to communicate with the kernel module.

Additionally, the library exposes two utility functions, via its header, libsessionfs.h: `get_sess_path()`, `write_sess_path()`, to change the folder in which session are enabled without making the userspace application communicate directly with the `SessionFS_dev` device. The last utility function exported is `device_shutdown()` which asks the device to disable itself and unlock the module.

The library is composed by:

- `__attribute__((constructor)) init_method()`: constructor used to save the libc `open` and `close`;
- `open()`: wraps the libc `open` function;
- `close()`: wraps the libc `close` function;
- `get_sess_path()`: returns the current session path;
- `write_sess_path()`: changes the current session path;
- `device_shutdown()`: ask to the device to disable itself and unlock the module.

The userspace library is not srictly necessary, but makes the communication with the `SessionFS_dev` device almost transparent from the perspective of the userspace program.

1.3 Kernel-Level Implementation

At kernel level there are several modules to implement the subsystem and several data structures to keep information about the opened sessions, which will be described in this section, referencing their documentation.

The subsystem is divided in four submodules:

- *Module Configuration*: Contains the kernel module configuration.
- *Character Device*: Contains the `SessionFS_dev` device implementation.
- *Session Manager*: Implements operations on the sessions.
- *Session Information*: Publishes sessions information under `SessionFS_dev` device kobject via SysFS.

1.3.1 Kernel-Level Data Structures Specification

The data structures are described following the dependencies of each submodule, starting from the *Module Configuration* submodule.

1.3.1.1 Module Configuration

On the kernel module there is only a module parameter, which is the session path of the device `sess_path`, to offer an alternative path for reading it.

1.3.1.2 Character Device

There are some global variables in the *Character Device* submodule, used to store information on the device. Then there are two headers, since we need to separate the information available for the kernel module, contained in device_sessionfs_mod.h, from the device information, which is available to the shared library, contained in device_sessionfs.h. In device_sessionfs.h the `sess_params` struct is used to pass parameters between userspace and kernel during ioctls.

It's important to note that in the device implementation (device_sessionfs.c) there are two global atomic variables:

- `device_status`: which determines if the device is disabled and it being removed.
- `refcount`: which keeps the information on the number of processes that are actually using the device.

As global variables there are also the session path (`sess_path`), the read-write lock for the session path (`dev_lock`) and the length of the string stored in `sess_path` as `path_len`.

1.3.1.3 Session Manager

As the *Character Device*, the *Session Manager* has two headers, one which exports its APIs, session_manager.h, and another, session_types.h, which contains the definition of the objects used by the session manager.

The objects used by this submodule are:

- `session_rcu`: Contains a pointer to a `session` object and is used in the rculist `sessions`.
- `session`: Represents an original file and its incarnations.
- `incarnation`: An incarnation file, related to an original file, so to a `session` object.
- `sess_info`: Used to publish informations on a struct `session` via SysFS.

The *Session Manager* implementation (session_manager.c), has also two global variables: an RCU list called `sessions`, which contains all the information about all the sessions and it's protected by a spinlock, `sessions_lock`, the other global variable.

1.3.1.4 Session Information

This submodule is the one responsible to publish information in the `SessionFS_dev` folder via SysFS, it exposes its APIs to the *Session Manager* submodule via session_info.h and uses the data structures defined in session_types.h among some global variables used in session_info.c:

- `sessions_num`: The variable holding the number of active sessions.
- `dev_kobj`: The `SessionFS_dev` kernel object, provided during the initialization of this subsystem.
- `kattr`: The kernel attribute of `SessionFS_dev` which represents the number of active sessions.

1.3.2 Kernel-Level Subsystem Implementation

This section contains the description about the implementation on the functions in each submodule, referencing their documentation, where the description of their implementation is more accurate.

1.3.2.1 Module Configuration

The *Module Configuration* (module.c) contains only two functions, along with the macros used to add `sess_path` as a read-only module paramenter.

The functions used in this module are:

- `sessionFS_load()`: which initializes the `SessionFS_dev` device and is used during the module initialization, as specified by `module_init()`.
- `sessionFS_unload()`: which releases the `SessionFS_dev` device and is used during the module exit, as specified by `module_exit()`.

The module exit function cannot be called anytime, because the module will be locked when in use by the `SessionFS_dev` device which adds a dependency on it.

1.3.2.2 Character Device

The *Character Device* submodule (device_sessionfs.c) implements the `SessionFS_dev` device and it's invoked by the *Module Configuration* submodule when the module is loaded, where `init_device()` is called and during the module unloading, where `release_device()` is called. These two functions are responsible of the device initialization and the device cleanup. It's important to note that when the device is initialized the kernel module is locked, using `try_get_module()`, which adds a dependency to the the kernel module, locking it.

When the `SessionFS_dev` device has been initialized it can be called from userspace applications, by issuing a read, write or ioctl operation.

When a read operation is issued the `device_read()` function is called, and the content of `sess_path` is copied in the provided buffer.

When a write operation is issued the `device_write()` is called and, after checking that the provided path is an absolute one, the `sess_path` buffer is reset and overwritten with the new content, also updating the `path_len` variable.

When an ioctl operation is issued the `device_ioctl()` is called and, according to the parameters, a session can be created with `create_session()`, closed with `close_session()` or a device shutdown can be requested.

If a device shutdown is requested the device checks if the dependency introduced during the `init_device()` should be removed; this happens only if the `refcount` is 0 and the *Session Manager* submodule has no sessions that are active or in use by SysFS.

1.3.2.3 Session Manager

The *Session Manager* submodule (session_manager.c) is initialized by the *Character Device* submodule, when `init_manager()` is called.

Then the *Character Device* can call `create_session()` or `close_session()` according the parameters passed in the received ioctl.

When `create_session()` is called, a `session` with the given `pathname` is searched (with `search_session()`) and if nothing valid is found a new object is created with `init_session()`. Then the `incarnation` object will be created with `create_incarnation()`. The original file is opened during the `session` creation, without a file descriptor associated. The incarnation file is opened during the `incarnation` creation with an associated file descriptor which will be used by the userspace program. `open_file()` is used to open both files.

When `close_session()` is called, the parent `session` of the `incarnation` is located using `search_session()`, then the `incarnation` is deleted using `delete_incarnation()`. The original file is overwritten only if the parent `session` is a valid one. If, after removing the `incarnation`, the parent `session` is empty and unused, it will be marked as invalid and removed as soon as possible.

This module also contains a primitive "garbage collector", the `clean_manager()` function, which will walk though all the sessions and will delete the `incarnation` associated with a process which is dead or in a zombie state, returning the number of sessions associated with an alive process. This function is used in the *Device Module* to determine if there are active sessions and thus if the module should remain locked even if no process is using the device at the moment.

The deletion of the `incarnation` objects is not immediate, since after removing its info on SysFS there could be processes with a reference on the related pseudofiles. So the object is marked as invalid and will be deallcoated along with the corresponding session object, when this is not in use.

1.3.2.4 Session Information

The *Session Information* (session_info.c) handles the publishing of the information on the session via SysFS, so it needs to be initialized with the kernel object of the `SessionFS_dev` device, using `init_info()`. Conversely, when the device is being released the kernel attribute used to publish the active sessions (`kattr`) needs to be removed from the kernel object of the device, using `release_info()`.

When an `session` is created by the *Session Manager* submodule, the function `add_session_info()` is called, which creates a new subfolder in the `SessionFS_dev` device folder and adds an `active_↩ incarnations_num` pseudofile in it. Conversely when a session is removed, the function `remove_incarnation_info()` is called removing the pseudofile and the folder.

Similarly, when an `incarnation` is created by the *Session Manager* submodule the function `add_incarnation_info()` is called, which creates a new pseudofile in the parent `session`, that has as filename the pid of the process which owns the `incarnation`, and gives the process name when read. Then the parent `session` `inc_num` variable, contained in the `sess_info` struct associated with the `session` object, is incremented. This variable is accessed when the `session` `active_incarnations_num` file is read. Conversely when an incarnation is removed, the function `remove_incarnation_info()` is called removing the pseudofile and decrementing the parent `session` `inc_num` variable.

1.4 Testcase and Benchmark

The module was tested on the latest long-term support kernel (which is 5.4 at the moment) and can be loaded and unloaded freely. However there could be situations in which the module cannot be unloaded, this is due to the fact the device is in use or there are sessions which are being used by active processes.

In particular there is a situation in which there are no active session and the module could be unloadable, this happens because the device requires an explicit ioctl to clean up the *Session Manager* submodule and unlock the module, if possible.0

The userspace program (userspace_test.c) is composed of four utility functions and the main, which will use them to carry out tests:

- `change_sess_path()`: changes the session path using the utility function provided by the shared userspace library.
- `func_test()`: tests the module functionalities in a per-process perspective.
- `fork_test()`: tests that the file semantic using session is the same when forking with opened sessions.
- `main()`: executes the tests for a use-case defined by the parameters passed when running the program.

  To use the program two parameters are required, in the following order:

  1. the maximum number of processes;
  2. the maximum number of files used by each process;

Tests have been conducted on four different use-cases, each of these use-cases is available to be used in the makefile under the `test` directory:

1. A single process which can open up to 15 files using sessions.
2. A single process which can open up to 100 file using sessions.
3. Up to 15 processes, where each of them can open up to 15 file using sessions.
4. Up to 100 processes, where each of them can open up to 100 file using sessions.

The single process use case are very quick and straightforward and were generally used to locate errors which made the module unusable. Instead the multi process test were designed to test how test how the module responds when there are many processes that use it and were used to locate errors in critical sections.

Tests were conducted initially in a virtualized enviroment on a kernel with debug features enabled, which has provided a mean to debug errors swiftly, without having to restart the machine each time there was a crash. When everything was functional on the virtual machine tests have been moved on the live environment. In particular we can see, especially in the use-case 4, that stress tests are lengthy since the processes can randomly sleep and they write a great amount of data to disk, in the use-case 4 more than 2GB can be written on disk. Tests completed successfully without errors and the kernel module responded swiftly and in the correct way when invoked. By examining the files we can see that the semantic of the Unix sessions is respected, since there are file that are completely overwritten and we can also see the interleaving of pids when processes append their content instead of overwriting.

1.5 Makefile organization

In the repository there are several makefiles, that execute operations with several levels of detail and where the "outer" makefiles user the "inner" makefile targets to execute its operations:

- In the root directory of the repository the makefile has only three targets:
  - `src`: which builds the source code for the kernel module, the shared library and the userspace program.
  - `docs`: which generates the documentation (including this report) in html and LaTeX in `docs/doxygen`.
  - `clean`: which will clean the repository from all the files that are generated by the above targets, excluding the pdf and html formats of the documentation.
- In the `src` directory the makefile has four targets:
  - `module`: which will build the kernel module.
  - `shared-lib`: which will build the userspace shared library.
  - `demo-lib`: which will build the userspace program.
  - `clean`: which will clean the repository from all the files that are generated by the above targets.
- In the `kmodule` directory the makefile has four targets:
  - `all`: which will build the kernel module.
  - `insmod`: which will run `dmesg -C` and insert the kernel module in the kernel with `insmod`.
  - `rmmod`: which will unload the module and kill `dmesg` if it is running.
  - `clean`: which will clean the repository from all the files that are generated by the above targets.
- In the `shared_lib` folder the makefile has only three targets:
  - `libsessionfs`: which will build the shared library.
  - `libsessionfs-d`: which will build the shared library enabling address sanitizer and the gdb debug options.
  - `clean`: which will clean the repository from all the files that are generated by the above target.
- In the `demo` folder the makefile has three targets:
  - `demo-lib-d`: which will compile,the userspace program enabling address sanitizer and the gdb debug options.
  - `demo-lib`: which will compile the userspace program linking it with the userspace shared library.
  - `clean`: which will clean the repository from all the files that are generated by the above target.
- In the `test` folder there are five targets:
  - `lib-test-single`: which will execute the use-case 1.
  - `lib-stress-test-single`: which will execute the use-case 2.
  - `lib-test-multi`: which will execute the use-case 3.
  - `lib-stress-test-multi`: which will execute the use-case 4.
  - `clean`: which will clean the repository from all the files that are generated by the above target, excluding log files.

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 incarnation Struct Reference

Informations on an incarnation of a file.

```
#include <session_types.h>
```

Data Fields

- struct file ∗ **file**
- int **filedes**
- struct kobj_attribute **inc_attr**
- struct llist_node **node**
- pid_t **owner_pid**
- const char ∗ **pathname**
- int **status**

4.1.1 Detailed Description

Informations on an incarnation of a file.

Parameters

node	Next `incarnation` on the list.
file	The struct file that represents the incarantion file.
inc_attr	a kernel object attribute that is used to read `incarnation` `owner_pid` and the process name.
pathname	The pathanme of the incarnation file.
filedes	File descriptor of the incarnation file.
owner_pid	Pid of the process that has requested the `incarnation`.
status	Contains the error code that could have invalidated the `incarnation`. If its value is less than 0 then the incarnation is invalid and must be closed as soon as possible.

This struct represents an incarnation file and it refers a `session` struct.

The documentation for this struct was generated from the following file:

- src/kmodule/session_types.h

4.2 sess_info Struct Reference

Infromations on a `session` used by SysFS.

```
#include <session_types.h>
```

Data Fields

- char ∗ **f_name**
- atomic_t **inc_num**
- struct kobj_attribute **inc_num_attr**
- struct kobject ∗ **kobj**

4.2.1 Detailed Description

Infromations on a `session` used by SysFS.

Parameters

kobj	The `session` kernel object.
inc_num_attr	The kernel object attribute that represents the number of incarnations for the original file.
f_name	Formatted filename of the session object, where each '/' is replaced by a '-'.
inc_num	The actual number of open incarnations for the original file.

This struct represents the published information about a `session`.

The documentation for this struct was generated from the following file:

- src/kmodule/session_types.h

4.3 sess_params Struct Reference

```
#include <device_sessionfs.h>
```

Data Fields

- int **filedes**
- int **flags**
- mode_t **mode**
- const char ∗ **orig_path**
- pid_t **pid**
- int **valid**

4.3.1 Detailed Description

Parameters

orig_path	The pathname of the original file to be opened in a session, or that represents the original file containing the incarnation to be closed.
flags	The flags used to determine the incarnation permissions.
mode	The permissions to apply to newly created files.
pid	The pid of the process that requests the creation of an incarnation.
filedes	The file descriptor of the incarnation.
valid	The session can be invalid if there was an error in the copying of the original file over the incarnation file, so the value of this parameter can be <= `VALID_SESS`.

This struct will hold all the necessary parameters used to open and close sessions.

The documentation for this struct was generated from the following file:

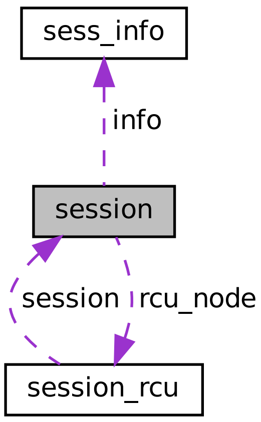
- src/kmodule/device_sessionfs.h

4.4 session Struct Reference

General information on a `session`.

```
#include <session_types.h>
```

Collaboration diagram for session:



Data Fields

- struct file ∗ **file**
- struct llist_head **incarnations**
- struct sess_info **info**
- const char ∗ **pathname**
- struct session_rcu ∗ **rcu_node**
- atomic_t **refcount**
- rwlock_t **sess_lock**
- atomic_t **valid**

4.4.1 Detailed Description

General information on a `session`.

Parameters

incarnations	List (lockless) of the active `incarnation`(s) of the file.
info	Informations on the current original file, represented by a<tt>::sess_info struct.
file	The struct file that represents the original file.
rcu_node	Pointer to the `session_rcu` that contains the current session object.
pathname	Pathname of the file that is opened with session semantic.
sess_lock	read-write lock used to ensure serialization in the session closures.
filedes	Descriptor of the file opened with session semantic.
refcount	The number of processes that are currently using this `session`.
valid	This parameter is used (after having gained the rwlock) to check if this struct `session` is still attached to the rculist.

This struct represent an original file with its active `incarnation`(s). If the session object has been removed from the rculist the value of this parameter will be different from `VALID_NODE`.

The documentation for this struct was generated from the following file:

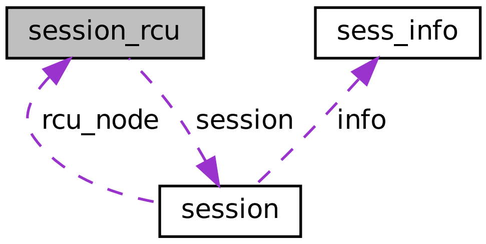
- src/kmodule/session_types.h

4.5 session_rcu Struct Reference

RCU item that contains a `session`.

```
#include <session_types.h>
```

Collaboration diagram for session_rcu:



Data Fields

- struct list_head **list_node**
- struct rcu_head **rcu_head**
- struct session ∗ **session**

4.5.1 Detailed Description

RCU item that contains a `session`.

Parameters

list_node	Used to navigate the list of sessions.
rcu_head	The rcu head structure used to protect the list with RCU.
session	The struct `session` which holds the session information.

We use this object because otherwise we couldn't determine if a process was using a struct session or simply walking the list.

The documentation for this struct was generated from the following file:

- src/kmodule/session_types.h

Chapter 5

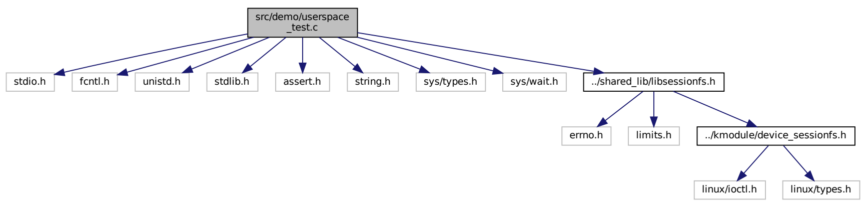
File Documentation

5.1 src/demo/userspace_test.c File Reference

Userpsace program that will test the kernel module using the libsessionfs.c shared library.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "../shared_lib/libsessionfs.h"
```

Include dependency graph for userspace_test.c:



Macros

- #define DEFAULT_PERM 0644

  Permissions to be used when calling `open()`

Functions

- int change_sess_path (char ∗path)

    Use libsessionfs APIs to change the session path to `path`.
- void fork_test (void)

    Test file semantic with a session opened when forking.
- void func_test (int files_max, char ∗base_fname)

    A general functionality test.
- int main (int argc, char ∗∗argv)

    Testing of the kernel module.
- void sess_change_test (void)

    Test the semantic of sessions when the session path has changed.

5.1.1 Detailed Description

Userpsace program that will test the kernel module using the libsessionfs.c shared library.

It will "simulate" a general use-case in which the module can be used, by operating on a pseudorandom number of files with a pseudorandom number of processes.

5.1.2 Function Documentation

5.1.2.1 change_sess_path()

```
int change_sess_path (
             char * path )
```

Use libsessionfs APIs to change the session path to `path`.

Parameters

in	*path*	The path in which the session path should be changed.

Returns

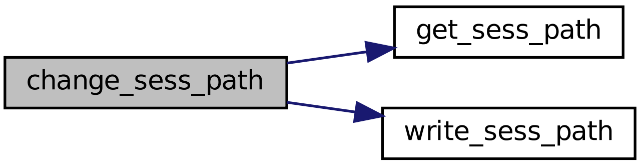
0 or -1 in case of error. (`errno` is set by libsessionfs functions).

This utility function uses `get_sess_path()` and `write_sess_path()` to read the current session path, change it and display the results to the user, testing the session path change feature.

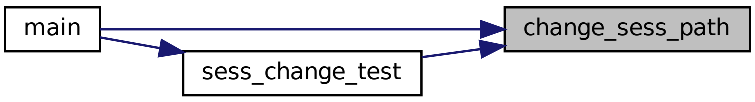
References get_sess_path(), and write_sess_path().

Referenced by main(), and sess_change_test().

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.2.2 fork_test()

```
void fork_test (
            void  )
```

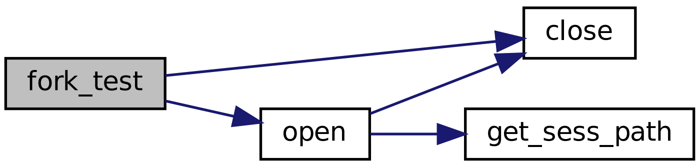
Test file semantic with a session opened when forking.

We open a file called `fork_test.txt` with session semantic then we execute the write test without appending done in `func_test()` on this file with both child and father processes. This is done to see if using session preserves the original semantic of the `read`, `write` and `llseek` functions. We can expect that one of the two processes will fail some operations on the file, since they have an intentional race condition on closing.

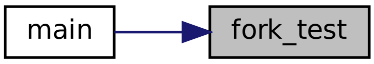
References close(), DEFAULT_PERM, O_SESS, and open().

Referenced by main().

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.2.3 func_test()

```
void func_test (
             int files_max,
             char * base_fname )
```

A general functionality test.

Parameters

in	*files_max*	The number of files to be used during the test.
in	*base_fname*	The string used to begin the filename of all the used files.

This function will test that all the features of the module are functional by simulating the common usage pattern that a single process could have, for a random number of files that goes from 0 to `files_max`. In detail we execute the following operations for each file:

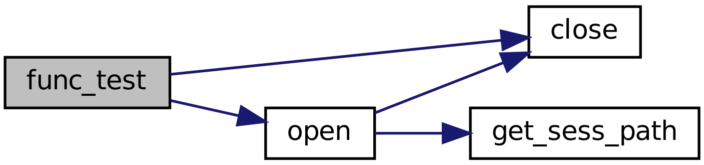
- read the `active_sessions_num` pseudofile;
- we can sleep at random for 1 second, to give the other process, if any, a chance to override the incarnation;
- open the file with the `O_SESS` flag;
- we check `active_sessions_num` pseudofile;
- we read `active_incarnations_num` and the file with our pid for each session we have created to ensure that they return meaningful values;
- we test `write`, `read` and `lseek` writing a random number of bytes from 0 to 1MB, by writing the pid serveral times:
  - before writing we decide at random to overwrite the file or to append our contents, to have a chance to see different pids in the original file;
  - we write the process pid;
  - we seek back to where we have written the last pid;
  - we read what we have written;
  - we check that there aren't mismatches;
- we seek to the beginning, middle and end of the file;

- we sleep for 1 second at random, to have some files born from the same incarnation in the concurrent test;
- we can close the opened file or leave it open at random to test how the module handles session with a dead owner;
- we check `active_sessions_num` pseudofile;

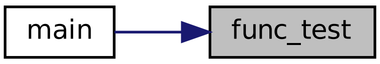
References close(), DEFAULT_PERM, O_SESS, and open().

Referenced by main().

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.2.4 main()

```
int main (
            int argc,
            char ** argv )
```

Testing of the kernel module.

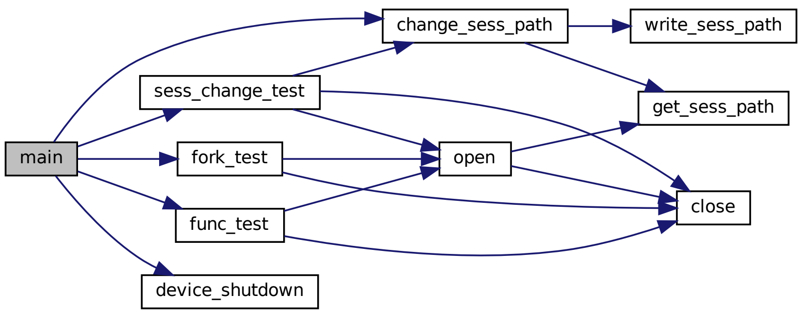
Parameters

in	*argc*	Number of the given arguments, 3 is expected.
in	*argv*	The arguments given to the file; we expect two arguments, the maximum number of processes to be used in the test followed by the maximum number of files to be used by each process.

We spawn a number of processes from 1 to the number given as first parameter, then in each child process we change the session path to the current directory using `change_sess_path()`, we execute `func_test()`, `sess_change_test()` and `fork_test()`. If the maximum number of processes used is 1 then all the files created by `func_test()` have a filename that starts with the `single_process` string, otherwise with the `multi_process` string. To be able remove the kernel module we need to power down the `SessionFS_dev` device, using the dedicated ioctl as the last operation on the device.

References change_sess_path(), device_shutdown(), fork_test(), func_test(), and sess_change_test().

Here is the call graph for this function:



5.1.2.5 sess_change_test()

```
void sess_change_test (
             void  )
```

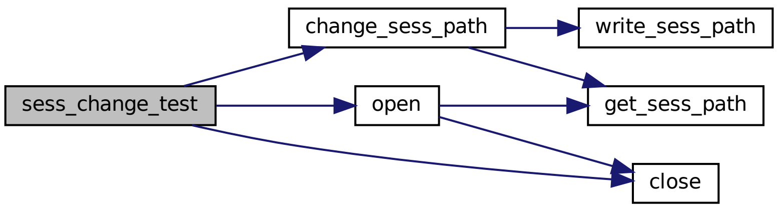
Test the semantic of sessions when the session path has changed.

We change the session path to the current directory, then we open a file with `O_SESS`, in the current directory. Afterwards we change the session path to `/mnt` and we open a file with `O_SESS` again in the current directory. Then we close both files. The filename of the files created starts with the `sess_change_test` string.

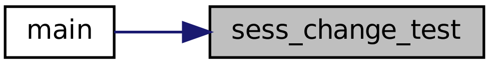
References change_sess_path(), close(), DEFAULT_PERM, O_SESS, and open().

Referenced by main().

Here is the call graph for this function:



Here is the caller graph for this function:

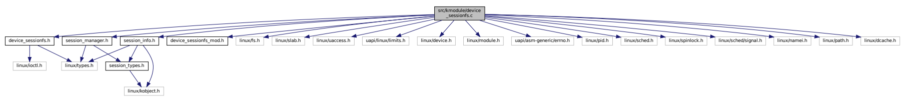


5.2 src/kmodule/device_sessionfs.c File Reference

Implementation of the virtual character device that will handle sessions, component of the *Character Device* sub-module.

```
#include "device_sessionfs.h"
#include "device_sessionfs_mod.h"
#include "session_manager.h"
#include "session_info.h"
#include <linux/fs.h>
#include <linux/slab.h>
#include <linux/uaccess.h>
#include <uapi/linux/limits.h>
#include <linux/device.h>
#include <linux/module.h>
#include <uapi/asm-generic/errno.h>
#include <linux/pid.h>
#include <linux/sched.h>
#include <linux/spinlock.h>
#include <linux/sched/signal.h>
#include <linux/namei.h>
#include <linux/path.h>
#include <linux/dcache.h>
```

Include dependency graph for device_sessionfs.c:



Macros

- #define DEFAULT_SESS_PATH "/mnt"

  The default session path when the device is initialized.
- #define DEVICE_DISABLED 1

  Indicates that the device has been disabled and is being removed.
- #define PATH_OK 1

  Indicates that the given path is contained in `sess_path`

Functions

- long int device_ioctl (struct file ∗file, unsigned int num, unsigned long param)

  Handles the ioctls calls issued to the `SessionFS_dev` *device.*
- static ssize_t device_read (struct file ∗file, char ∗buffer, size_t buflen, loff_t ∗offset)

  Get the path in which sessions are enabled.
- static ssize_t device_write (struct file ∗file, const char ∗buffer, size_t buflen, loff_t ∗offset)

  Writes a new path in which sessions must be enabled.
- int init_device (void)

  Device initialization and registration.
- int path_check (const char ∗path)

  Check if the given path is a subpath of `sess_path`
- void release_device (void)

  Releases the device and frees the used memory.
- static char ∗ sessionfs_devnode (struct device ∗dev, umode_t ∗mode)

  Allows every user to read and write the device file of our virtual device.

Variables

- struct device ∗ dev =NULL

  Device object.
- struct class ∗ dev_class =NULL

  Device class.
- rwlock_t dev_lock

  Lock that protects the session path from concurrent accesses.
- struct file_operations ∗ dev_ops =NULL

  File operations allowed on our device.
- atomic_t device_status

  Indicates that the device must not be used since is being removed.
- int path_len =0

  Length of the of the path string.
- atomic_t refcount

  Refcount of the processes that are using the device.
- char ∗ sess_path =NULL

  The path to the directory in which session sematic is enabled.

5.2.1 Detailed Description

Implementation of the virtual character device that will handle sessions, component of the *Character Device* sub-module.

This file contains the implementation of the device operations that will handle the session semantics and the interaction with the path where sessions are enabled.

5.2.2 Function Documentation

5.2.2.1 device_ioctl()

```
long int device_ioctl (
            struct file * file,
            unsigned int num,
            unsigned long param )
```

Handles the ioctls calls issued to the `SessionFS_dev` device.

Parameters

`in`	*file*	The special file that represents our char device.
`in`	*num*	The ioctl sequence number, used to identify the operation to be executed, its possible values are `IOCTL_SEQ_OPEN`, `IOCTL_SEQ_CLOSE` and `IOCTL_SEQ_SHUTDOWN`.
`in,out`	*param*	The ioctl param, which is a `sess_params` struct, that contains the information on the session that must be opened/closed and will be updated with the information on the result of the operation.

Returns

0 on success or an error code. (`-ENODEV` if the device is disabled, `-EINVAL` if `path_check()` fails or parameters are invalid, `-EAGAIN` if the `copy_to_user` fails and `-EPIPE` plus a `SIGPIPE` signal if the original file can't be found.)

This function checks `device_status`, increments `refcount`, then copies the `sess_params` struct and its `orig_pathname` in kernel space. Its behaviour differs in base of the ioctl sequence number specified:

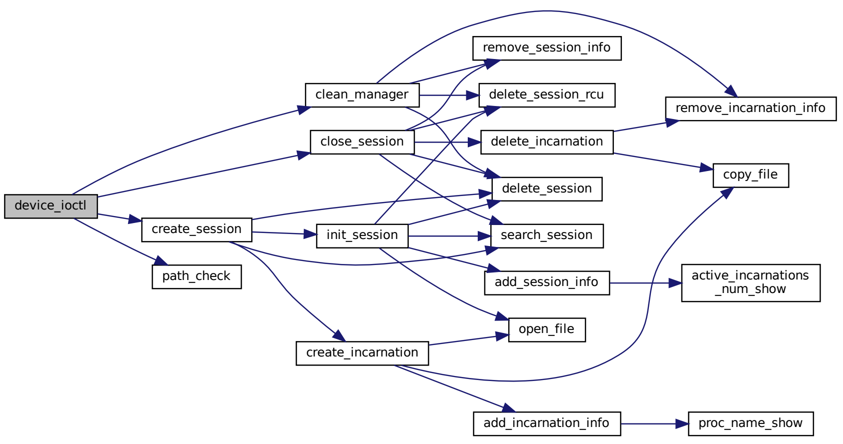
- `IOCTL_SEQ_OPEN`: Tries to create a session by invoking `create_session()`. If the session and the incarnation are created successfully the file descriptor of the incarnation is copied into `sess_params` `filedes` member. If the incarnation gets corrupted during creation, the `filedes` member of `sess_params` is updated as in the successful case, but the corresponding error code is returned, so that the library can close and remove the corrupted incarnation file.
- `IOCTL_SEQ_CLOSE`: closes an open session using `close_session()` and the incarnation file must be closed and removed by the library. If the original file does not exist anymore it sends `SIGPIPE` to the user process.
- `IOCTL_SEQ_SHUTDOWN`: disables the device, setting `device_status` to `DEVICE_DISABLED`, to avoid race conditions. Then calls `clean_manager()` to check if there are active sessions. If there are no active sessions and the refcount is 1 (we are the only process using the device) then the module is unlocked, using `module_put()`. Otherwise the device is re-enabled, setting `device_status` to `!DEVICE_DISABLED` and the ioctl fails with `-EAGAIN`.

To allow the unload of the module we need to have no active sessions, no processes that are using the device and no processes that are using the device kobject.

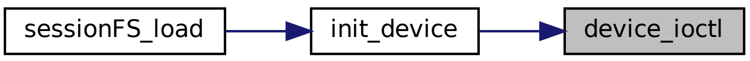
References clean_manager(), close_session(), create_session(), dev, DEVICE_DISABLED, device_status, IOCTL_SEQ_CLOSE, IOCTL_SEQ_OPEN, IOCTL_SEQ_SHUTDOWN, O_SESS, path_check(), PATH_OK, refcount, and sess_path.

Referenced by init_device().

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.2.2 device_read()

```
static ssize_t device_read (
              struct file * file,
              char * buffer,
              size_t buflen,
              loff_t * offset )  [static]
```

Get the path in which sessions are enabled.

Parameters

out	*buffer*	The buffer in which the path is copied.
in	*buflen*	The lenght of the supplied buffer.
	file	unused, but necessary to fit the function into struct file_operations.
	offset	unused, but necessary to fit the function into struct file_operations.

Returns

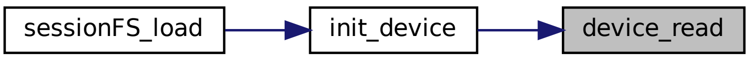
The number of bytes written in `buffer`, or an error code (`-EINVAL` if one of the supplied parameters is invalid, `-EAGAIN` if the `copy_to_user()` failed, `-ENODEV` if the device is disabled).

This function will copy the `sess_path` content in the supplied buffer. The first operations be executed are the check for the device status on `device_status`, the incrementation of the `refcount`. Then `dev_lock` is grabbed for reading; after the operation is completed `dev_lock` is released and the `refcount` is decremented.

References dev_lock, DEVICE_DISABLED, device_status, path_len, refcount, and sess_path.

Referenced by init_device().

Here is the caller graph for this function:



5.2.2.3 device_write()

```
static ssize_t device_write (
            struct file * file,
            const char * buffer,
            size_t buflen,
            loff_t * offset )  [static]
```

Writes a new path in which sessions must be enabled.

Parameters

in	*buffer*	The new path in which session are enabled.
in	*buflen*	The length of the provided buffer.
	file	unused, but necessary to fit the function into struct file_operations.
	offset	unused, but necessary to fit the function into struct file_operations.

Returns

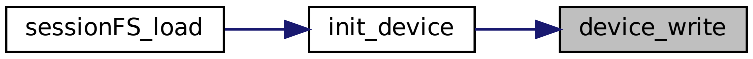
The number of bytes written in `buffer`, or an error code (`-EINVAL` if one of the supplied parameters are invalid, `-EAGAIN` if the copy_from_user failed, `-ENODEV` if the device is disabled).

This function will reset and overwrite `sess_path`, without affecting existing sessions, if the supplied path is absolute. To do so we check `device_status` and we increment `refcount`. Then we check that the supplied path starts with '/', grab `dev_lock` for write operations, we zero-fill `sess_path`, copy the new string and add a terminator, just in case. Finally we release `dev_lock` and decrement `refcount`.

References dev_lock, DEVICE_DISABLED, device_status, path_len, refcount, and sess_path.

Referenced by init_device().

Here is the caller graph for this function:



5.2.2.4 init_device()

```
int init_device (
             void   )
```

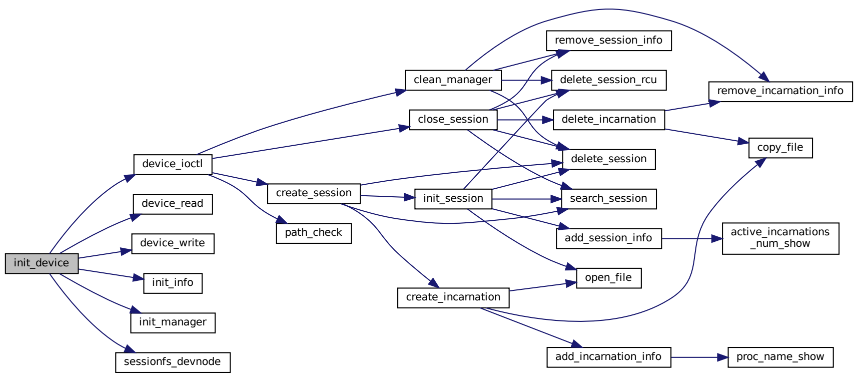
Device initialization and registration.

Initializes and registers the device by setting `sess_path`, `path_len` variables: `dev_ops` will contain the operations allowed on the device, which are `device_ioctl()`, `device_read()` and `device_write()`, and the `sessionfs_devnode()` callback to set the inode permissions. The *Session Manager* submodule is also initialized using `init_manager()` and the same happens for the *Session Information* submodule, using `init_info()`, after the device is registered. Finally we lock the module with `try_module_get()` to prevent it being unmounted while is in use.

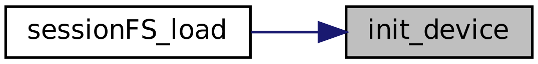
References CLASS_NAME, DEFAULT_SESS_PATH, dev, dev_class, dev_lock, dev_ops, DEVICE_DISABLED, device_ioctl(), DEVICE_NAME, device_read(), device_status, device_write(), init_info(), init_manager(), MAJOR_NUM, path_len, refcount, sess_path, and sessionfs_devnode().

Referenced by sessionFS_load().

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.2.5 path_check()

```
int path_check (
             const char * path )
```

Check if the given path is a subpath of `sess_path`

Gets the dentry from the given path and from `sess_path` and checks if the second dentry is an ancestor of the first dentry.

Parameters

`in`	*path*	Path to be checked

Returns

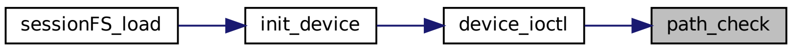
`PATH_OK` if the given path is a subpath of `sess_path` and !`PATH_OK` otherwise; an error code is returned on error.

If the dentry corresponding to the given path cannot be found, the function will check if `sess_path` is a substring of the given path.

References dev_lock, PATH_OK, and sess_path.

Referenced by device_ioctl().

Here is the caller graph for this function:



5.2.2.6 release_device()

```
void release_device (
             void  )
```

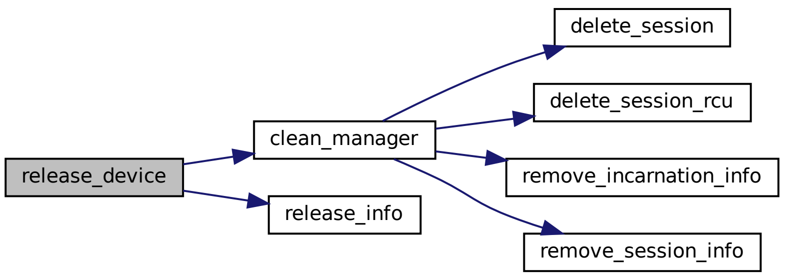
Releases the device and frees the used memory.

Unregisters the device, cleans the *Session Manager* just to be sure to avoid memory leaks, releases the *Session Information* and frees the used memory (`dev_ops` and `sess_path`).

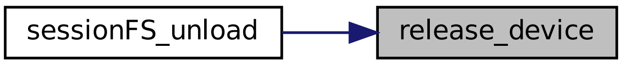
References clean_manager(), dev_class, dev_ops, DEVICE_DISABLED, DEVICE_NAME, device_status, MAJOR_NUM, release_info(), and sess_path.

Referenced by sessionFS_unload().

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.2.7 sessionfs_devnode()

```
static char* sessionfs_devnode (
             struct device * dev,
             umode_t * mode )  [static]
```

Allows every user to read and write the device file of our virtual device.

Parameters

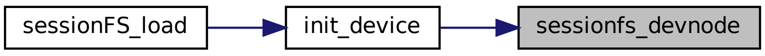
in	*dev*	Our device struct.
out	*mode*	The permissions we set to our device.

With this callback that is added to `dev_ops` we set the permissions of the inode in `/dev` that represents our device, allowing every user to read and write in it.

References dev, and MAJOR_NUM.

Referenced by init_device().

Here is the caller graph for this function:



5.2.3 Variable Documentation

5.2.3.1 sess_path

```
char* sess_path =NULL
```

The path to the directory in which session sematic is enabled.

Keeps the path to the directory in which session sematic is enabled (located in ::device_sessionfs.c).

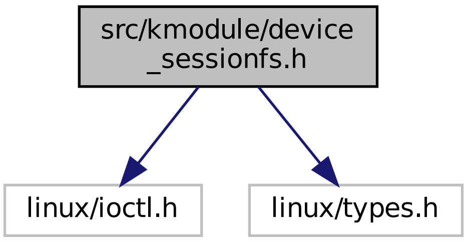
Referenced by close(), device_ioctl(), device_read(), device_write(), init_device(), open(), path_check(), and release_device().

5.3 src/kmodule/device_sessionfs.h File Reference

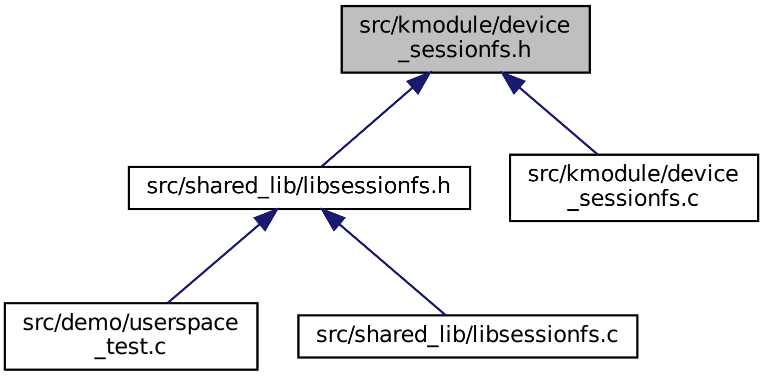
Properties of the devices necessary for the device implementation and the library that uses the device, component of the *Character Device* submodule.

```
#include <linux/ioctl.h>
#include <linux/types.h>
```

Include dependency graph for device_sessionfs.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct sess_params

Macros

- #define CLASS_NAME "SessionFS_class"

  The name of the corresponding device class.
- #define DEVICE_NAME "SessionFS_dev"

  The name of our virtual device.
- #define IOCTL_CLOSE_SESSION _IOWR(MAJOR_NUM,IOCTL_SEQ_CLOSE,struct sess_params∗)

  We define the ioctl command for closing a session.
- #define IOCTL_DEVICE_SHUTDOWN _IOR(MAJOR_NUM,IOCTL_SEQ_SHUTDOWN,int∗)

We define the ioctl command fot asking a device shutdown.
- #define IOCTL_OPEN_SESSION _IOWR(MAJOR_NUM,IOCTL_SEQ_OPEN,struct sess_params∗)

  We define the ioctl command for opening a session.
- #define IOCTL_SEQ_CLOSE 1

  The ioctl sequence number that idenfies the closing of a session.
- #define IOCTL_SEQ_OPEN 0

  The ioctl sequence number that indenfies the opening of a session.
- #define IOCTL_SEQ_SHUTDOWN 10

  The ioctl sequence number that idenfies the request for the device shutdown.
- #define MAJOR_NUM 120
- #define O_SESS 10000000

  Flag used to enable the Unix session semantic.
- #define VALID_SESS 0

  Defines the validity of a session.

5.3.1 Detailed Description

Properties of the devices necessary for the device implementation and the library that uses the device, component of the *Character Device* submodule.

This file contains some general device properties, such as the available ioctls and the device name that are needed to the file that will implement the device and to the library the will use the ioctls.

5.3.2 Macro Definition Documentation

5.3.2.1 IOCTL_CLOSE_SESSION

```
#define IOCTL_CLOSE_SESSION _IOWR(MAJOR_NUM,IOCTL_SEQ_CLOSE,struct sess_params*)
```

We define the ioctl command for closing a session.

We use the macro `_IOWR` since we need to pass to the virtual device the `sess_params` struct.

5.3.2.2 IOCTL_DEVICE_SHUTDOWN

```
#define IOCTL_DEVICE_SHUTDOWN _IOR(MAJOR_NUM,IOCTL_SEQ_SHUTDOWN,int*)
```

We define the ioctl command fot asking a device shutdown.

We use the `_IOR` macro since the device will let the userspace program read the number of active sessions during shutdown.

5.3.2.3 IOCTL_OPEN_SESSION

```
#define IOCTL_OPEN_SESSION _IOWR(MAJOR_NUM,IOCTL_SEQ_OPEN,struct sess_params*)
```

We define the ioctl command for opening a session.

We use the `_IOWR` macro since we need to pass to the virtual device the `sess_params` struct.

5.3.2.4 MAJOR_NUM

```
#define MAJOR_NUM 120
```

A major device number is necessary to identify our virtual device, since it doensn't have an assigned letter. We use 120 as major number since it reserved for local and experimental use. See: Documentation/admin-guide/devices.txt

5.3.2.5 O_SESS

```
#define O_SESS 10000000
```

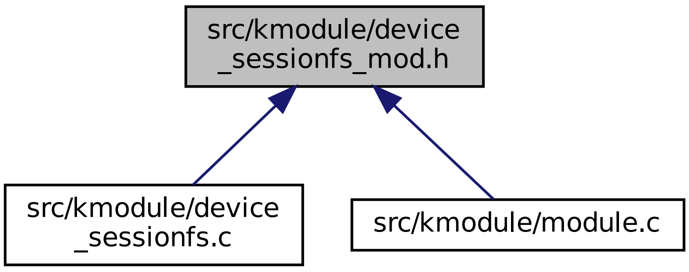
Flag used to enable the Unix session semantic.

Unused flag in `include/uapi/asm-generic/fcntl.h`, that has been repurposed.

5.4 src/kmodule/device_sessionfs_mod.h File Reference

Device properties needed to the kernel module, component of the *Character Device* submodule.

This graph shows which files directly or indirectly include this file:


src/kmodule/device_sessionfs_mod.h
src/kmodule/device_sessionfs.c
src/kmodule/module.c

Functions

- int init_device (void)

    Device initialization and registration.
- void release_device (void)

    Releases the device and frees the used memory.

Variables

- char ∗ sess_path

  Keeps the path to the directory in which session sematic is enabled (located in ::device_sessionfs.c).

5.4.1 Detailed Description

Device properties needed to the kernel module, component of the *Character Device* submodule.

This file contains the properties of the device that are needed to the kernel module, such as the prototypes of the device init and cleanup functions and the buffer that will contain the path in which sessions are enabled.

5.4.2 Function Documentation

5.4.2.1 init_device()

```
int init_device (
             void  )
```

Device initialization and registration.

Returns

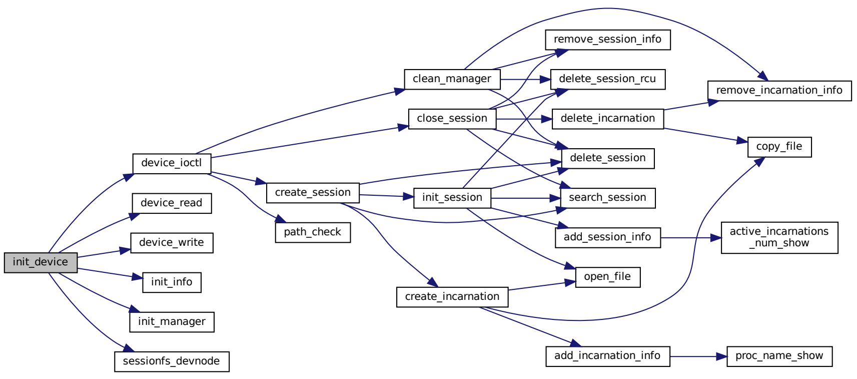
0 on success -1 on error.

Initializes and registers the device by setting `sess_path`, `path_len` variables: `dev_ops` will contain the operations allowed on the device, which are `device_ioctl()`, `device_read()` and `device_write()`, and the `sessionfs_devnode()` callback to set the inode permissions. The *Session Manager* submodule is also initialized using `init_manager()` and the same happens for the *Session Information* submodule, using `init_info()`, after the device is registered. Finally we lock the module with `try_module_get()` to prevent it being unmounted while is in use.

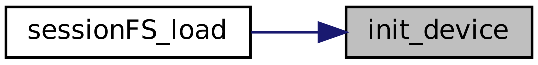
References CLASS_NAME, DEFAULT_SESS_PATH, dev, dev_class, dev_lock, dev_ops, DEVICE_DISABLED, device_ioctl(), DEVICE_NAME, device_read(), device_status, device_write(), init_info(), init_manager(), MAJOR_NUM, path_len, refcount, sess_path, and sessionfs_devnode().

Referenced by sessionFS_load().

Here is the call graph for this function:



Here is the caller graph for this function:



5.4.2.2 release_device()

```
void release_device (
             void  )
```

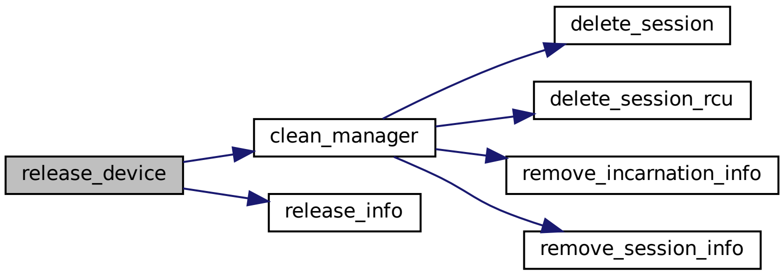
Releases the device and frees the used memory.

Unregisters the device, cleans the *Session Manager* just to be sure to avoid memory leaks, releases the *Session Information* and frees the used memory (`dev_ops` and `sess_path`).

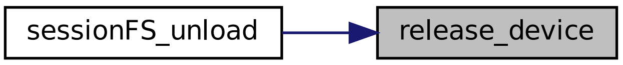
References clean_manager(), dev_class, dev_ops, DEVICE_DISABLED, DEVICE_NAME, device_status, MAJOR_NUM, release_info(), and sess_path.

Referenced by sessionFS_unload().

Here is the call graph for this function:



Here is the caller graph for this function:



5.4.3 Variable Documentation

5.4.3.1 sess_path

```
char* sess_path
```

Keeps the path to the directory in which session sematic is enabled (located in ::device_sessionfs.c).

Keeps the path to the directory in which session sematic is enabled (located in ::device_sessionfs.c).

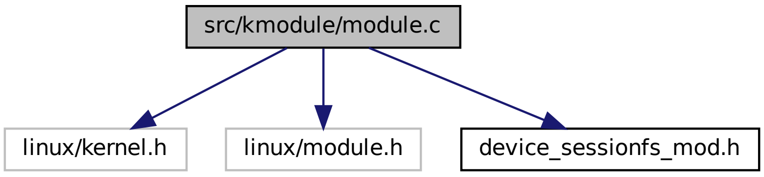
Referenced by close(), device_ioctl(), device_read(), device_write(), init_device(), open(), path_check(), and release_device().

5.5 src/kmodule/module.c File Reference

Module configuration, component of the *Module Configuration* submodule.

```
#include <linux/kernel.h>
#include <linux/module.h>
#include "device_sessionfs_mod.h"
```

Include dependency graph for module.c:



Functions

- MODULE_AUTHOR ("Mattia Nicolella <mattianicolella@gmail.com>")

  Module author specification.
- MODULE_DESCRIPTION ("A session based virtual filesystem wrapper")

  A short description of the module.
- module_exit (sessionFS_unload)

  Specification of the module cleanup function.
- module_init (sessionFS_load)

  Specification of the module init function.
- MODULE_LICENSE ("GPL")

  Specification of the license used by the module. Close sourced module cannot access to all the kernel facilities. This is intended to avoid open source code to be stolen by closed source developers.
- module_param (sess_path, charp, 0444)

  We set the session path as a read-only module parameter.
- **MODULE_PARM_DESC** (sess_path,"path in which session sematic is enabled")
- MODULE_VERSION ("0.5")

  Module version specification.
- static int __init sessionFS_load (void)

  Loads the device when the kernel module is loaded in the kernel.
- static void __exit sessionFS_unload (void)

5.5.1 Detailed Description

Module configuration, component of the *Module Configuration* submodule.

This file contains the module configuration and the functions that will be executed when the module is loaded and unloaded.

5.5.2 Function Documentation

5.5.2.1 sessionFS_load()

```
static int __init sessionFS_load (
            void )  [static]
```

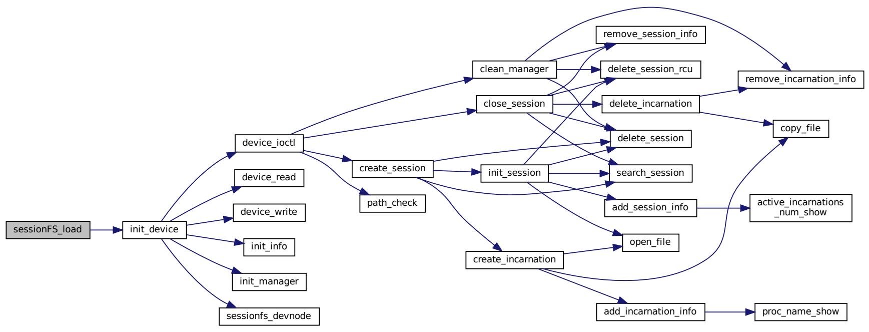
Loads the device when the kernel module is loaded in the kernel.

Returns

0 on success, and error code on fail

References init_device().

Here is the call graph for this function:



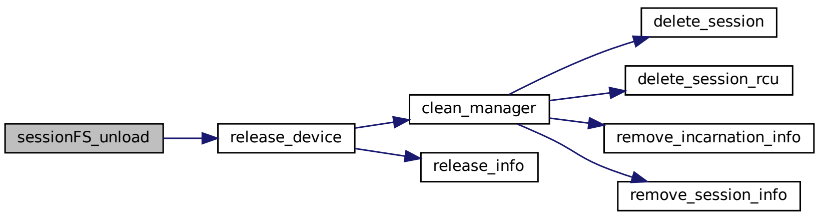
5.5.2.2 sessionFS_unload()

```
static void __exit sessionFS_unload (
             void  )  [static]
```

Before unloading the module we relase the device.

References release_device().

Here is the call graph for this function:

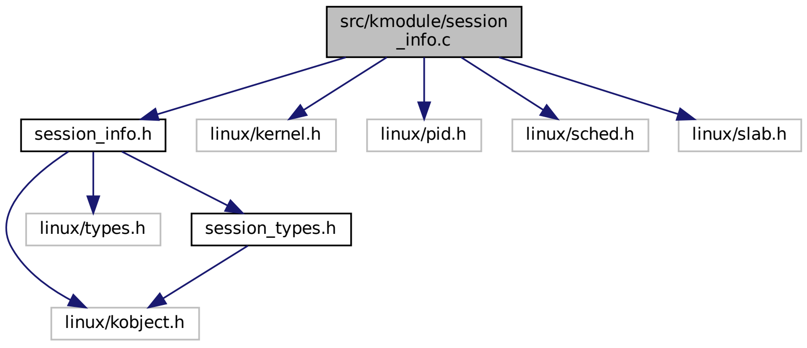


5.6 src/kmodule/session_info.c File Reference

Implementation of the *Session Information* submodule.

```
#include "session_info.h"
#include <linux/kernel.h>
#include <linux/pid.h>
#include <linux/sched.h>
#include <linux/slab.h>
```

Include dependency graph for session_info.c:



Macros

- #define KERN_OBJ_PERM 0444

    Kernel objects attributes are read only, since we only read information on sessions.

Functions

- ssize_t active_incarnations_num_show (struct kobject ∗obj, struct kobj_attribute ∗attr, char ∗buf)

    The function used to read the SysFS `active_incarnations_num` attribute file.
- ssize_t active_sessions_num_show (struct kobject ∗obj, struct kobj_attribute ∗attr, char ∗buf)

    The function used to read the SysFS `active_sessions_num` attribute file.
- int add_incarnation_info (struct sess_info ∗parent_session, struct kobj_attribute ∗incarnation, pid_t pid, int fdes)

    Adds a new kobject attribute representing an incarnation.
- int add_session_info (const char ∗name, struct sess_info ∗session)

    Adds a new kobject representing an original file, under the SessionFS kobject.
- int init_info (struct kobject ∗device_kobj)

    Initializes the SessionFS kobject with general information about the running sessions.
- ssize_t proc_name_show (struct kobject ∗obj, struct kobj_attribute ∗attr, char ∗buf)

    The function used to read the SysFS incarnations attribute files.
- void release_info (void)

    Removes the SessionFS information from the device provided in `init_info()`.
- void remove_incarnation_info (struct sess_info ∗parent_session, struct kobj_attribute ∗incarnation)

    Removes the kobject attribute incarnation from a kobject.
- void remove_session_info (struct sess_info ∗session)

    Removes a session kobject.

Variables

- struct kobject ∗ dev_kobj

    The device kobject provided during `init_info()`.
- struct kobj_attribute kattr = __ATTR_RO(active_sessions_num)

    The kernel attribute that will contain the number of open sessions.
- atomic_t sessions_num

    The number of opened sessions.

5.6.1 Detailed Description

Implementation of the *Session Information* submodule.

5.6.2 Function Documentation

5.6.2.1 active_incarnations_num_show()

```
ssize_t active_incarnations_num_show (
            struct kobject * obj,
            struct kobj_attribute * attr,
            char * buf )
```

The function used to read the SysFS `active_incarnations_num` attribute file.

Parameters

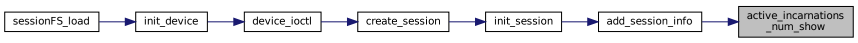
in	*obj*	The kobject that has the attribute being read.
in	*attr*	The attribute of the kobject that is being read.
out	*buf*	The buffer (which is PAGE_SIZE bytes long) that contains the file contents.

Returns

The number of bytes read (in [0,PAGE_SIZE]). The file content returned is the number of active incarnations for the current original file.

Referenced by add_session_info().

Here is the caller graph for this function:



5.6.2.2 active_sessions_num_show()

```
ssize_t active_sessions_num_show (
             struct kobject * obj,
             struct kobj_attribute * attr,
             char * buf )
```

The function used to read the SysFS `active_sessions_num` attribute file.

Parameters

in	*obj*	The kobject that has the attribute being read.
in	*attr*	The attribute of the kobject that is being read.
out	*buf*	The buffer (which is PAGE_SIZE bytes long) that contains the file contents.

Returns

The number of bytes read (in [0,PAGE_SIZE]). The file content is the number of active sessions.

References sessions_num.

5.6.2.3 add_incarnation_info()

```
int add_incarnation_info (
             struct sess_info * parent_session,
             struct kobj_attribute * incarnation,
             pid_t pid,
             int fdes )
```

Adds a new kobject attribute representing an incarnation.

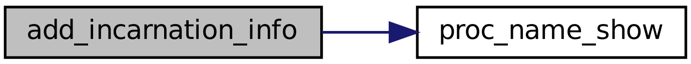
By adding a new incarnation we increment `active_sessions_num` and `active_incarnations_num` for the given `session`, represented by its `sess_info` member, also, a kobject attribute is added to the given `session` that has the process pid as filename and contains the process name. Finally the reference counter of the given `session` is also incremented.

The format of the file found on SysFS `[pid]_[file descriptor]`, to allow the same process to have the process open the same file multiple times.

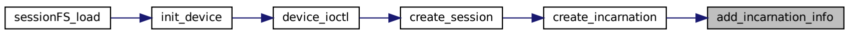
References KERN_OBJ_PERM, proc_name_show(), and sessions_num.

Referenced by create_incarnation().

Here is the call graph for this function:



Here is the caller graph for this function:



5.6.2.4 add_session_info()

```
int add_session_info (
            const char * name,
            struct sess_info * session )
```

Adds a new kobject representing an original file, under the SessionFS kobject.

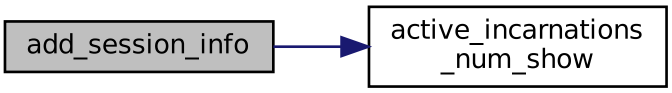
We add a new folder in sysfs which is represented by the given kobject. The `session` kobject, represented by the `sess_info` member the `session` object will be created as a child of `dev_kobj`, and the `dev_kobj` reference counter will be incremented. We also format the filename substituting '/' with '-'.

Finally, initialize the number of incarnations as a kobj_attribute.

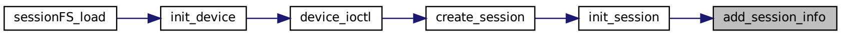
References active_incarnations_num_show(), dev_kobj, and KERN_OBJ_PERM.

Referenced by init_session().

Here is the call graph for this function:



Here is the caller graph for this function:



5.6.2.5 init_info()

```
int init_info (
            struct kobject * device_kobj )
```

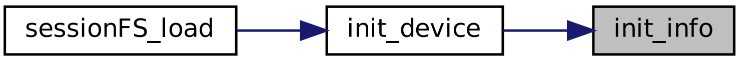
Initializes the SessionFS kobject with general information about the running sessions.

We add an attribute called `active_sessions_num` to the SessionFS device kernel object, which is only readable and its content is the number of active sessions.

References dev_kobj, kattr, and sessions_num.

Referenced by init_device().

Here is the caller graph for this function:



5.6.2.6 proc_name_show()

```
ssize_t proc_name_show (
            struct kobject * obj,
            struct kobj_attribute * attr,
            char * buf )
```

The function used to read the SysFS incarnations attribute files.

Parameters

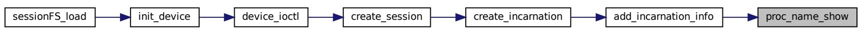
`in`	*obj*	The kobject that has the attribute being read.
`in`	*attr*	The attribute of the kobject that is being read.
`out`	*buf*	The buffer (which is PAGE_SIZE bytes long) that contains the file contents.

Returns

The number of bytes read (in [0,PAGE_SIZE]). The file content is the process name that corresponds to the pid used as filename.

Referenced by add_incarnation_info().

Here is the caller graph for this function:



5.6.2.7 release_info()

```
void release_info (
            void )
```

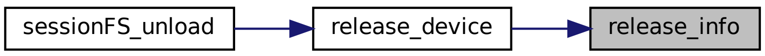
Removes the SessionFS information from the device provided in `init_info()`.

We remove the 'active_sessions_num' attribute from the device

References dev_kobj, and kattr.

Referenced by release_device().

Here is the caller graph for this function:



5.6.2.8 remove_incarnation_info()

```
void remove_incarnation_info (
             struct sess_info * parent_session,
             struct kobj_attribute * incarnation )
```

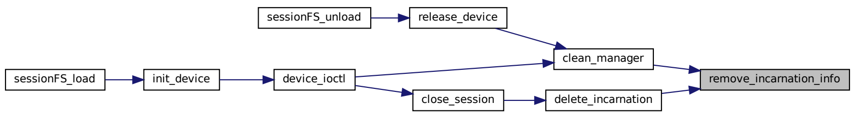
Removes the kobject attribute incarnation from a kobject.

By removing an incarnation we decrement `active_sessions_num` and `active_incarnations_num` for the given `session`, represented by its `sess_info` member. Also, the kobject attribute that has the process pid as filename and contains the process name is removed from the given `session`. Finally the reference counter of the given `session` is also decremented.

References sessions_num.

Referenced by clean_manager(), and delete_incarnation().

Here is the caller graph for this function:



5.6.2.9 remove_session_info()

```
void remove_session_info (
             struct sess_info * session )
```

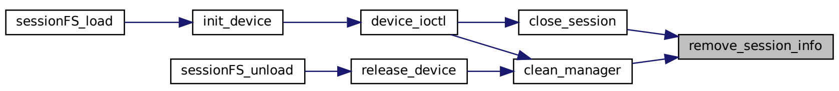
Removes a session kobject.

Removes the entry corresponding to the given `session`, represented by its `sess_info` member, in the device SysFS folder. To do so we also remove the `active_incarnations_num` file of the given `session` and we decrement the reference counter of the device session kernel object.

References dev_kobj.

Referenced by clean_manager(), and close_session().

Here is the caller graph for this function:

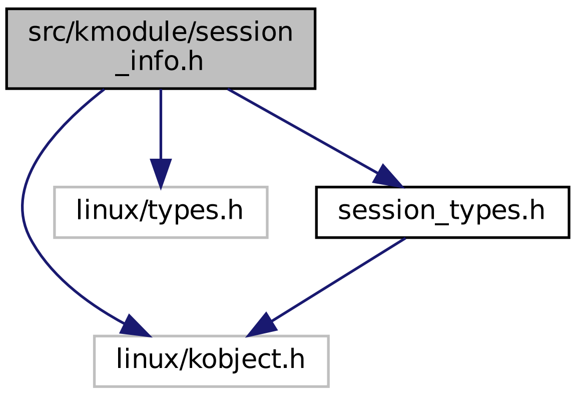


5.7 src/kmodule/session_info.h File Reference

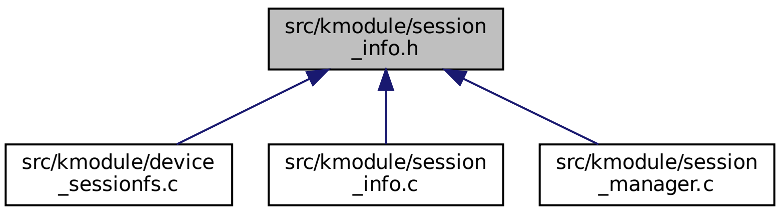
Handles the kobjects for the session manager, part of the *Session Information* submodule. This file is used to create and manage kobjects for the session manager.

```
#include <linux/kobject.h>
#include <linux/types.h>
#include "session_types.h"
```

Include dependency graph for session_info.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define ATTR_GROUP_NAME "info"

  Each attribute group has the same name, but different attributes according to the parent kobject.
- #define SESS_KOBJ_NAME "SessionFS_info"

  The name of the Kobject representing the session manager in SysFS.

Functions

- int add_incarnation_info (struct sess_info ∗parent_session, struct kobj_attribute ∗incarnation, pid_t pid, int fdes)

  Adds a new kobject attribute representing an incarnation.
- int add_session_info (const char ∗name, struct sess_info ∗session)

  Adds a new kobject representing an original file, under the SessionFS kobject.
- int init_info (struct kobject ∗device_kobj)

  Initializes the SessionFS kobject with general information about the running sessions.
- void release_info (void)

  Removes the SessionFS information from the device provided in `init_info()`.
- void remove_incarnation_info (struct sess_info ∗parent_session, struct kobj_attribute ∗incarnation)

  Removes the kobject attribute incarnation from a kobject.
- void remove_session_info (struct sess_info ∗session)

  Removes a session kobject.

5.7.1 Detailed Description

Handles the kobjects for the session manager, part of the *Session Information* submodule. This file is used to create and manage kobjects for the session manager.

5.7.2 Function Documentation

5.7.2.1 add_incarnation_info()

```
int add_incarnation_info (
              struct sess_info ∗ parent_session,
              struct kobj_attribute ∗ incarnation,
              pid_t pid,
              int fdes )
```

Adds a new kobject attribute representing an incarnation.

Parameters

in	*parent_session*	The session which has generated the incarnation, represented by a struct `sess_info`.
in	*incarnation*	The incarnation to be added.
in	*pid*	The pid of the process that owns the incarnation.
in	*fdes*	The file descriptor that identifies the incarnation in the process.

Returns

0 on success, or an error code.

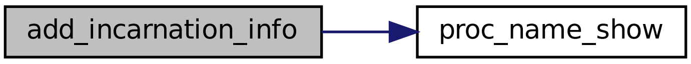
By adding a new incarnation we increment `active_sessions_num` and `active_incarnations_num` for the given `session`, represented by its `sess_info` member, also, a kobject attribute is added to the given `session` that has the process pid as filename and contains the process name. Finally the reference counter of the given `session` is also incremented.

The format of the file found on SysFS `[pid]_[file descriptor]`, to allow the same process to have the process open the same file multiple times.

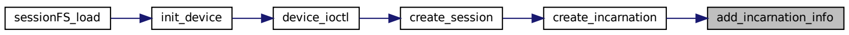
References KERN_OBJ_PERM, proc_name_show(), and sessions_num.

Referenced by create_incarnation().

Here is the call graph for this function:



Here is the caller graph for this function:



5.7.2.2 add_session_info()

```
int add_session_info (
             const char * name,
             struct sess_info * session )
```

Adds a new kobject representing an original file, under the SessionFS kobject.

Parameters

`in`	*name*	The name of the created kobject.
`in,out`	*session*	The information on the session, represente by a struct `sess_info`.

Returns

a struct kobject∗ on success, null if the kobject hasn't been created or an error code.

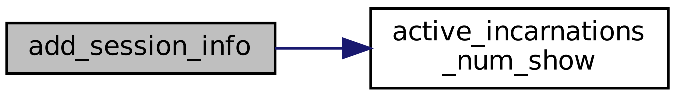
We add a new folder in sysfs which is represented by the given kobject. The `session` kobject, represented by the `sess_info` member the `session` object will be created as a child of `dev_kobj`, and the `dev_kobj` reference counter will be incremented. We also format the filename substituting '/' with '-'.

Finally, initialize the number of incarnations as a kobj_attribute.

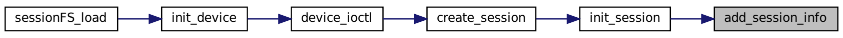
References active_incarnations_num_show(), dev_kobj, and KERN_OBJ_PERM.

Referenced by init_session().

Here is the call graph for this function:



Here is the caller graph for this function:



5.7.2.3 init_info()

```
int init_info (
            struct kobject * device_kobj )
```

Initializes the SessionFS kobject with general information about the running sessions.

Parameters

in	*device_kobj*	The SessionFS char device kernel object, in which contains the info on all sessions.

Returns

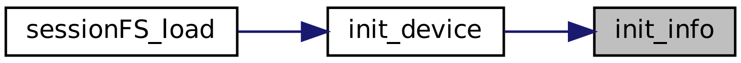
0 on success, an error code on failure.

We add an attribute called `active_sessions_num` to the SessionFS device kernel object, which is only readable and its content is the number of active sessions.

References dev_kobj, kattr, and sessions_num.

Referenced by init_device().

Here is the caller graph for this function:



5.7.2.4 release_info()

```
void release_info (
             void  )
```

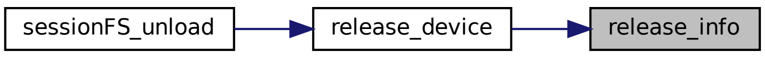
Removes the SessionFS information from the device provided in `init_info()`.

We remove the 'active_sessions_num' attribute from the device

References dev_kobj, and kattr.

Referenced by release_device().

Here is the caller graph for this function:



5.7.2.5 remove_incarnation_info()

```
void remove_incarnation_info (
             struct sess_info * parent_session,
             struct kobj_attribute * incarnation )
```

Removes the kobject attribute incarnation from a kobject.

Parameters

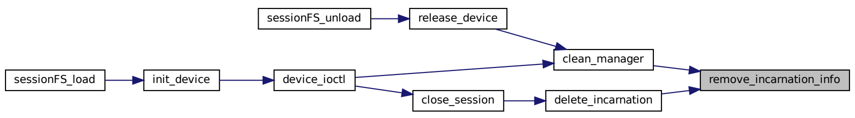
`in`	*parent_session*	The session which has generated the incarnation, represented by a struct `sess_info`.
`in`	*incarnation*	The incarnation to be added.

By removing an incarnation we decrement `active_sessions_num` and `active_incarnations_num` for the given `session`, represented by its `sess_info` member. Also, the kobject attribute that has the process pid as filename and contains the process name is removed from the given `session`. Finally the reference counter of the given `session` is also decremented.

References sessions_num.

Referenced by clean_manager(), and delete_incarnation().

Here is the caller graph for this function:



5.7.2.6 remove_session_info()

```
void remove_session_info (
            struct sess_info * session )
```

Removes a session kobject.

Parameters

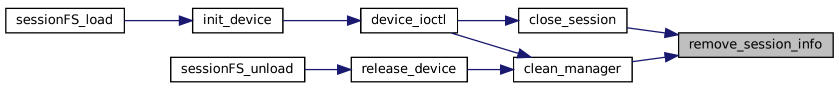
in	*session*	the information on the session to be removed, represented by a struct `sess_info`.

Removes the entry corresponding to the given `session`, represented by its `sess_info` member, in the device SysFS folder. To do so we also remove the `active_incarnations_num` file of the given `session` and we decrement the reference counter of the device session kernel object.

References dev_kobj.

Referenced by clean_manager(), and close_session().

Here is the caller graph for this function:

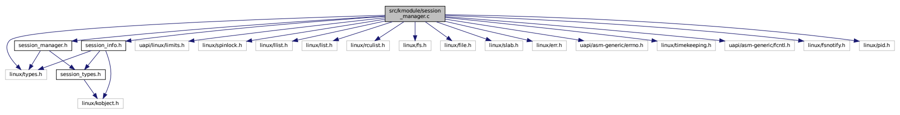


5.8 src/kmodule/session_manager.c File Reference

Implementation of the *Session Manager* submodule.

```
#include <linux/types.h>
#include <uapi/linux/limits.h>
#include <linux/spinlock.h>
#include <linux/llist.h>
#include <linux/list.h>
#include <linux/rculist.h>
#include <linux/fs.h>
#include <linux/file.h>
#include <linux/slab.h>
#include <linux/err.h>
#include <uapi/asm-generic/errno.h>
#include <linux/timekeeping.h>
#include <uapi/asm-generic/fcntl.h>
#include <linux/fsnotify.h>
#include <linux/pid.h>
#include "session_manager.h"
#include "session_info.h"
```

Include dependency graph for session_manager.c:



Macros

- #define DATA_DIM 512

  The portion of the file which is copied at each read/write iteration.
- #define DEFAULT_PERM 0644

  Permissions to be given to the newly created files.
- #define MANAGER_EMPTY 0

  Used to determine is the session manager contains active sessions.
- #define NO_FD 0

  Used to toggle the necessity of a file descriptor in `open_file()` and to determine the type of search in `search_session()`.
- #define NO_PID 0

  Used to determine the type of search in `search_session()`.
- #define OVERWRITE_ORIG 0

  Used to determine if the content of the incarnation must overwrite the original file on close.
- #define VALID_NODE 0

  Used to determine if a session node is valid.

Functions

- int clean_manager (void)

  Releases all the incarnations that are associated with a dead/zombie pid.
- int close_session (const char ∗pathname, int fdes, pid_t pid)

  Closes a session.
- int copy_file (struct file ∗src, struct file ∗dst)

  Copy the contents of a file into another.
- struct incarnation ∗ create_incarnation (struct session ∗session, int flags, pid_t pid, mode_t mode)

Creates an `incarnation` and add it to an existing `session`.
- struct incarnation ∗ create_session (const char ∗pathname, int flags, pid_t pid, mode_t mode)

  Create a new session for the specified file.
- int delete_incarnation (struct session ∗session, int filedes, pid_t pid, int overwrite)

  Removes the given `incarnation`.
- void delete_session (struct session ∗session)

  Deallocates the given session object.
- void delete_session_rcu (struct rcu_head ∗head)

  Deallocates a `session_rcu` element.
- int init_manager (void)

  Initialization of the session manager data structures.
- struct session ∗ init_session (const char ∗pathname, int flags, mode_t mode)

  Initializes the session information for the given pathname.
- int open_file (const char ∗pathname, int flags, mode_t mode, int fd_needed, struct file ∗∗file)

  Opens a file from kernel space.
- struct session ∗ search_session (const char ∗pathname, int filedes, pid_t pid)

  Searches for a `session` with a given pathname, or with an `incarnation` with matching pid and file descriptor.

Variables

- struct list_head sessions

  List of the active `session`(s).
- spinlock_t sessions_lock

  Spinlock used to update the list of the active `session`(s)

5.8.1 Detailed Description

Implementation of the *Session Manager* submodule.

This file contains the implementation of the session manager, which will handle the creation and deletion of a session, keeping track of the opened sessions for each file.

5.8.2 Function Documentation

5.8.2.1 clean_manager()

```
int clean_manager (
             void   )
```

Releases all the incarnations that are associated with a dead/zombie pid.

This method will walk through the `sessions` list and each `incarnation` list, deleting all the `incarnation`(s) and `session`(s) in which the process that has requested it is not active anymore, leaving the original files untouched. If there is an `incarnation` that is still in use by an active process it will be preserved.

To check if an `incarnation` is still active we get the struct pid from the pid number included in the `incarnation`; if we receive it, the process is at most in a zombie state.

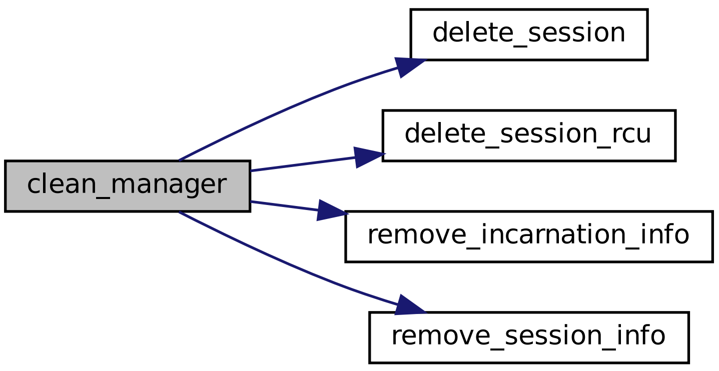
When walking the `sessions` list we increment the refcount of the `session` in use to avoid having it removed while we are using it. We remove all the elements in the `incarnations` lockless list and we add back the objects, closing the others without modifing the original file and updating SysFS with `remove_incarnation_info()`, without deallcoating the dead `incarnation`(s). If a `session` has no active incarnations it will be flagged as invalid and removed during a second walk, where, with `delete_incarnation()`, we remove all the invalid `session`(s) and deallocate al the associated `incarnation`(s).

NOTE: For dead incarnations that have not been closed we leave the files in the folder, since can't remove them from kernel space. Userspace will need to manually remove incarnations file for invalid processes. We consider as invalid a session which is associated to a dead process.

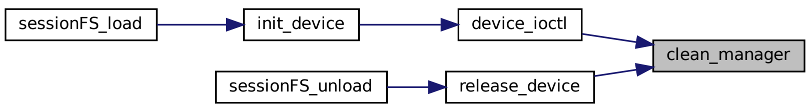
References delete_session(), delete_session_rcu(), MANAGER_EMPTY, remove_incarnation_info(), remove_session_info(), sessions, sessions_lock, and VALID_NODE.

Referenced by device_ioctl(), and release_device().

Here is the call graph for this function:



Here is the caller graph for this function:



5.8.2.2 close_session()

```
int close_session (
            const char * pathname,
            int fdes,
            pid_t pid )
```

Closes a session.

This function will close one session, by finding the corresponding `incarnation`, using `search_session()`, copying the incarnation file over the original file (atomically in respect to other session operations on the same original file, and only if the `session` is valid), and deleting the incarnation, using `delete_incarnation()`. If after the incarnation deletion the `session` has no other `incarnation`(s) the it will also schedule the `session` to be removed. If we cannot locate the parent session of an incarnation `-EBADF` is returned.

To remove a session object we need to check several conditions:

- The `session` must be not in use by other threads (refcount==1)
- The `session` kernel object refcount, in the `info` member, must be 1
- The `session` must be still valid and not already marked for deletion

If the current `session` must be removed, we flag it as invalid, to avoid having new incarnations created in here before deallocating it and making sure that it will be eventually deallocated.

We also remove its information on SysFS using `remove_session_info()`.

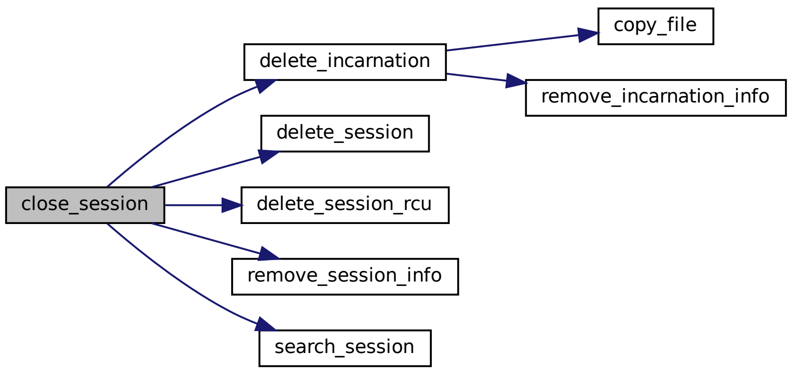
Then, we can remove the current `session` object from the rcu list, using the `sessions_lock` spinlock to avoid concurrent operations.

Finally, we try to deallocate the `session` if is invalid, using `delete_session()`.

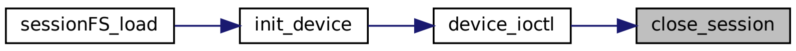
References delete_incarnation(), delete_session(), delete_session_rcu(), OVERWRITE_ORIG, remove_session_info(), search_session(), sessions_lock, and VALID_NODE.

Referenced by device_ioctl().

Here is the call graph for this function:



Here is the caller graph for this function:



5.8.2.3 copy_file()

```
int copy_file (
            struct file * src,
            struct file * dst )
```

Copy the contents of a file into another.

Parameters

in	*src*	The source file.
in	*dst*	The destination file.

Returns

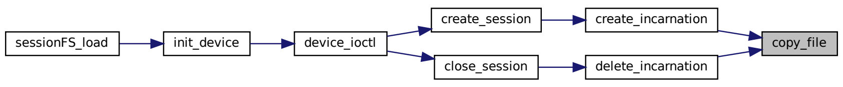
0 on success, an error code on failure.

Reads `DATA_DIM` bytes from `src` and writes them on `dst`, starting in both files from the beginning and stopping when `src` is has been completely read.

References DATA_DIM.

Referenced by create_incarnation(), and delete_incarnation().

Here is the caller graph for this function:



5.8.2.4 create_incarnation()

```
struct incarnation* create_incarnation (
            struct session * session,
            int flags,
            pid_t pid,
            mode_t mode )
```

Creates an `incarnation` and add it to an existing `session`.

Parameters

in	*session*	The session object that represents the file in which we want to create a new incarnation.
in	*flags*	The flags the regulates how the file must be opened.
in	*pid*	The pid of the process that wants the create a new incarnation.
in	*mode*	The permissions to apply to newly created files.

Returns

The file descriptor of the new `incarnation` or an error code (`-EAGAIN` if the parent session is invalid).

Creates an `incarnation` by updating the information on SysFS using `add_incarnation_info()` and opening a new file, using `open_file()`, copying the contents of the original file in the new file, using `copy_file()`.Then creates an `incarnation` object, filling it with info and adding it to the `incarnations` list of the parent `session`.

The original flags will be modified by adding the `O_CREAT` flag, since the incarnation file must always be created.

If the created incarnation is invalid the error code that has invalidated the session can be found in the `incarnation status` parameter.

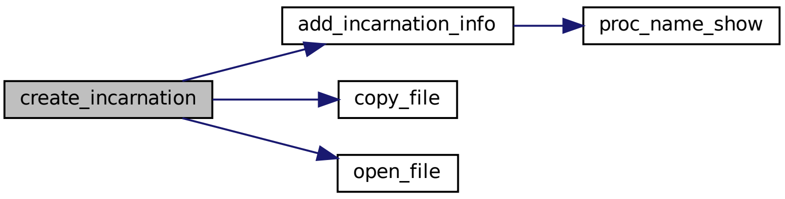
If we have an invalid parent `session` we return `-EAGAIN`.

The incarnation file name has the following format: `[original filename]_incarnation_[pid]_↩ [timestamp]` to make the incarnation file unique. If the original pathname is too long we use the following format `/var/tmp/[pid]_[timestamp]`. The timestamp is obtained by calling `ktime_get_real()`. We need to grab the parent `session` lock in read mode from when we copy the original file over the incarnation file; since we do not need to protect the lockless list when adding elements, but the `session` incarnations must be created atomically in respect to close operations on the same original file The lock is released when the incarnation has been added to the list.

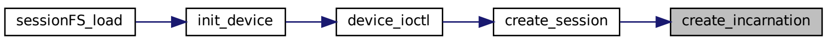
References add_incarnation_info(), copy_file(), NO_FD, open_file(), and VALID_NODE.

Referenced by create_session().

Here is the call graph for this function:



Here is the caller graph for this function:



5.8.2.5 create_session()

```
struct incarnation* create_session (
              const char * pathname,
              int flags,
              pid_t pid,
              mode_t mode )
```

Create a new session for the specified file.

To create a new session we check if the original file was already opened with session semantic, by searching for an existing `session` with the same `pathname` using `search_session()`. If the found `session` is invalid or a matching `session` object is not found a new `session` object will be created, using `init_session()`. Then we create a new `incarnation` of the original file with `create_incarnation()`.

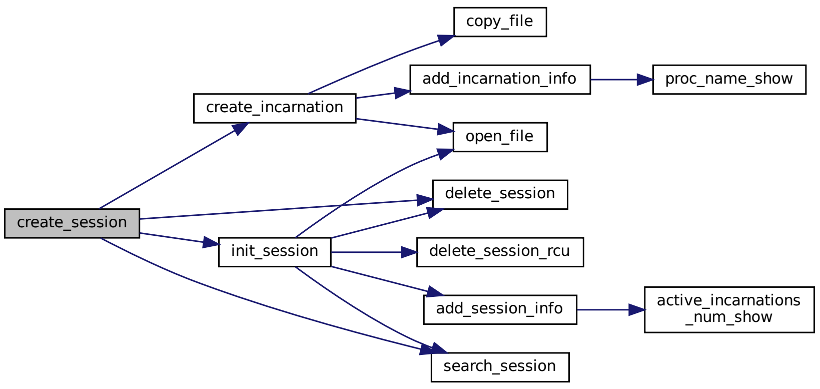
When the `incarnation` has been created the `refcount` of the parent session is decremented.

`-EAGAIN` is returned if the created session is invalid.

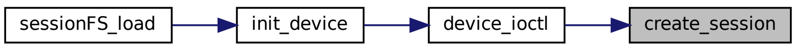
References create_incarnation(), delete_session(), init_session(), NO_FD, NO_PID, search_session(), and VALID_NODE.

Referenced by device_ioctl().

Here is the call graph for this function:



Here is the caller graph for this function:



5.8.2.6 delete_incarnation()

```
int delete_incarnation (
            struct session * session,
            int filedes,
            pid_t pid,
            int overwrite )
```

Removes the given `incarnation`.

Parameters

`in`	*session*	The session containing the `incarnation` to be removed.
`in`	*filedes*	The file descriptor that identifies the `incarnation`
`in`	*pid*	The pid of the owner of the `incarnation`.
`in`	*overwrite*	If set to `OVERWRITE_ORIG` it will overwrite the original file with the content of the `incarnation` which is going to be removed, otherwise the current `incarnation` is simply removed.

Returns

0 or an error code (`-ENOENT` if the incarnation can't be found).

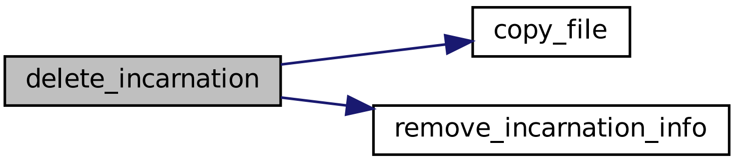
Searches the `incarnations` list of the given `session` for the current `incarnation`, copies the contents of the `incarnation` over the original file (if `overwrite` is set to `OVERWRITE_ORIG`) and marks it as invalid. The userspace library needs to close and remove the file.

When the incarnation is removed, SysFS is updated with `remove_incarnation_info()`. There could be processes still referencing the `incarnation` associated SysFS files after their removal, so the `incarnation` will remain in the lockless list and will be deallocated when the `session` will be deleted. The `incarnation` to be closed will be marked as invalid, by setting its `status` member to `-ENOENT`

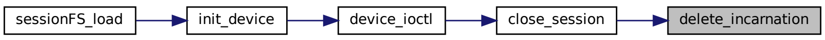
References copy_file(), OVERWRITE_ORIG, remove_incarnation_info(), and VALID_NODE.

Referenced by close_session().

Here is the call graph for this function:



Here is the caller graph for this function:



5.8.2.7 delete_session()

```
void delete_session (
             struct session * session )
```

Deallocates the given session object.

Parameters

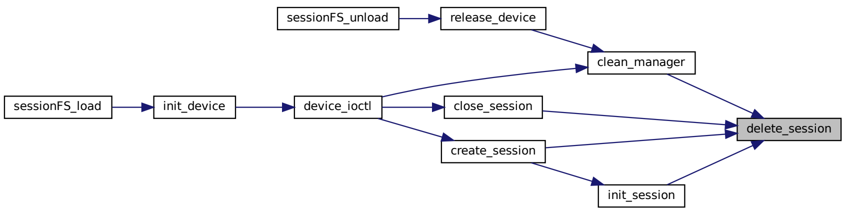
in	*session*	The session object to deallocate.

This function is used to free the memory used by the session when nobody is accessing it, this is checked using the session refcount and the session kernel object refcount, in the info member.

The method will attempt to deallocate the session object, with all its incarnations, if the session refcount is 0 and the associated kernel object refcount is 1 (only the current process is using it). Otherwise the method will do nothing.

Referenced by clean_manager(), close_session(), create_session(), and init_session().

Here is the caller graph for this function:



5.8.2.8 delete_session_rcu()

```
void delete_session_rcu (
             struct rcu_head * head )
```

Deallocates a session_rcu element.

Parameters

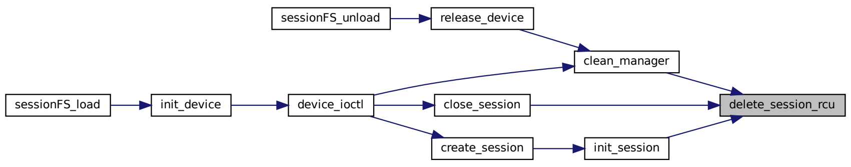
in	*head*	The rcu_head struct contained inside the session_rcu to deallcoate.

Used to deallocate the session_rcu element after it has been removed from the sessions list.

DO NOT directly call this function, instead you should let this function be called by `call_rcu()` when nobody is accessing anymore this element.

Referenced by clean_manager(), close_session(), and init_session().

Here is the caller graph for this function:



5.8.2.9 init_manager()

```
int init_manager (
            void )
```

Initialization of the session manager data structures.

Initializes the `sessions` global variable as an empty list. Avoids the RCU initialization since we can't receive requests yet, so no one will use this list for now. Then initializes the `sessions_lock` spinlock.

References sessions, and sessions_lock.

Referenced by init_device().

Here is the caller graph for this function:

sessionFS_load
init_device
init_manager

5.8.2.10 init_session()

```
struct session* init_session (
            const char * pathname,
            int flags,
            mode_t mode )
```

Initializes the session information for the given pathname.

Parameters

in	*pathname*	The path of the original file.
in	*flags*	The flags that regulate the access to the original file.
in	*mode*	The permissions to apply to newly created files.

To create a new `session` object we open the matching file using `open_file()`, then we get the spinlock `sessions_lock` to avoid race coditions and a search is issued, using `search_session()`, to see if there is already a matching session with the same `pathname` to be returned.

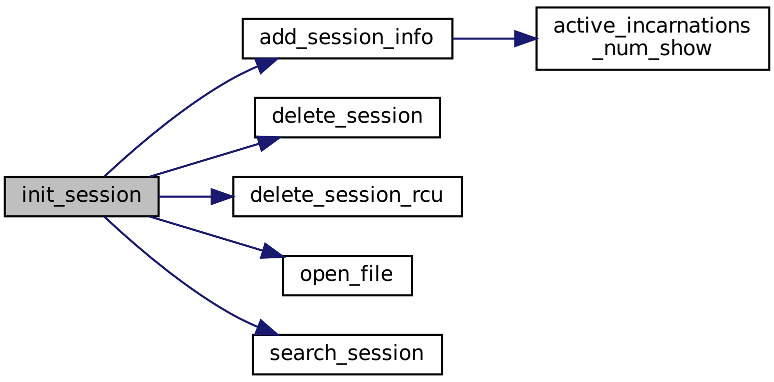
If the search gives an invalid object or `NULL` we proceed in the session creation, creating and adding the `session` object in the `sessions` list and calling `add_session_info()`. Finally we release the spinlock.

The original flags will be modified by removing the `O_RDONLY` and `O_WRONLY` in favor of `O_RDWR`, since we will always read and write on this file. In this way we preserve the effects of the `O_EXCL` flag if specified from userspace.

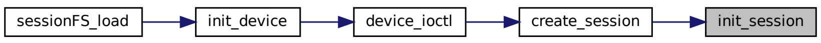
References add_session_info(), delete_session(), delete_session_rcu(), NO_FD, open_file(), search_session(), sessions, sessions_lock, and VALID_NODE.

Referenced by create_session().

Here is the call graph for this function:



Here is the caller graph for this function:



5.8.2.11 open_file()

```
int open_file (
            const char * pathname,
            int flags,
            mode_t mode,
            int fd_needed,
            struct file ** file )
```

Opens a file from kernel space.

Parameters

in	*pathname*	String that represents the file location and name and **must be in kernel memory**
in	*flags*	Flags that will regulate the permissions on the file.
in	*mode*	The permissions to apply to newly created files.
in	*fd_needed*	If set to NO_FD the file descriptor for the openend file will not be assigned (used only when we are opening original files).
out	*file*	pointer to the file struct of the opened file.

Returns

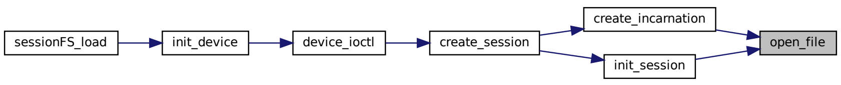
The file descriptor on success, the error code or NO_FD if no file descriptor was requested.

This function will open the file at the specified pathname and then associate a file descriptor to the opened file. If the O_CREAT flags is specified the permissions for the newly created file will be defined by the mode parameter or, if this is -1 by DEFAULT_PERM. If fd_needed is set to NO_FD then the opened file won't have a file descriptor associated.

References DEFAULT_PERM, and NO_FD.

Referenced by create_incarnation(), and init_session().

Here is the caller graph for this function:



5.8.2.12 search_session()

```
struct session* search_session (
            const char * pathname,
            int filedes,
            pid_t pid )
```

Searches for a session with a given pathname, or with an incarnation with matching pid and file descriptor.

Parameters

in	*pathname*	The pathname that identifies the session.
in	*filedes*	The file descriptor of an incarnation.
in	*pid*	The pid of the process that owns the incarnation.

Returns

A pointer to the found `session` or `NULL`.

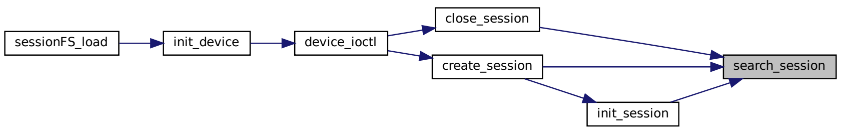
Searches, by navigating the `sessions` rcu list, a `session` which matches the given `pathname`. If `pathname` is NULL, `filedes` is diferrent from `NO_FD` and `pid` is diferrent from `NO_PID` then it searches for the session that contains an incarnation with the corresponding pid an file descriptor. While walking the `sessions` list the reference counter counter of any session currently inspected, `refcount`, will be incremented. If the session is not the one we are looking for then `refcount` will be decremented before inspecting the next `session`. If a `session` is invalid it will be skipped.

Additionally while walking the `incarnation` list of a `session` we grab the `sess_lock` in read mode, to avoid that a concurrent process modifies the list while we are walking it. When we finish walking the list we will release the lock.

References NO_FD, NO_PID, sessions, and VALID_NODE.

Referenced by close_session(), create_session(), and init_session().

Here is the caller graph for this function:

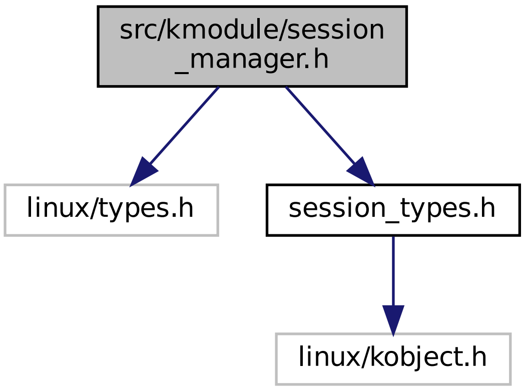


5.9 src/kmodule/session_manager.h File Reference

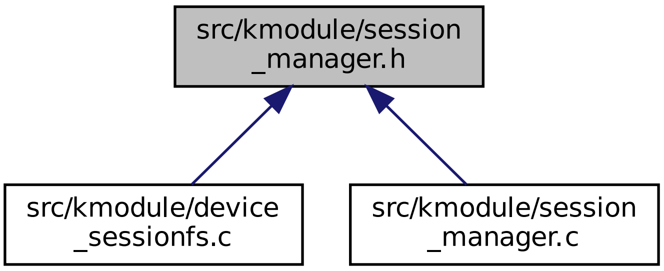
APIs used to interact with the session manger, component of the *Session Manager* submodule.

```
#include <linux/types.h>
#include "session_types.h"
```

Include dependency graph for session_manager.h:



This graph shows which files directly or indirectly include this file:



Functions

- int clean_manager (void)

  Releases all the incarnations that are associated with a dead/zombie pid.
- int close_session (const char ∗pathname, int fdes, pid_t pid)

  Closes a session.
- struct incarnation ∗ create_session (const char ∗pathname, int flags, pid_t pid, mode_t mode)

  Create a new session for the specified file.
- int init_manager (void)

  Initialization of the session manager data structures.

5.9.1 Detailed Description

APIs used to interact with the session manger, component of the *Session Manager* submodule.

This file contains the APIs of the session manager, which are used to initialize and release the manager and to add and remove sessions.

5.9.2 Function Documentation

5.9.2.1 clean_manager()

```
int clean_manager (
            void  )
```

Releases all the incarnations that are associated with a dead/zombie pid.

Returns

the number of sessions associated with an active pid.

This method will walk through the `sessions` list and each `incarnation` list, deleting all the `incarnation`(s) and `session`(s) in which the process that has requested it is not active anymore, leaving the original files untouched. If there is an `incarnation` that is still in use by an active process it will be preserved.

To check if an `incarnation` is still active we get the struct pid from the pid number included in the `incarnation`; if we receive it, the process is at most in a zombie state.

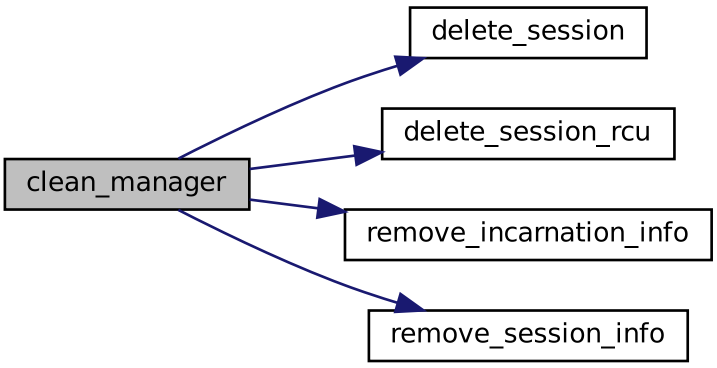
When walking the `sessions` list we increment the refcount of the `session` in use to avoid having it removed while we are using it. We remove all the elements in the `incarnations` lockless list and we add back the objects, closing the others without modifing the original file and updating SysFS with `remove_incarnation_info()`, without deallcoating the dead `incarnation`(s). If a `session` has no active incarnations it will be flagged as invalid and removed during a second walk, where, with `delete_incarnation()`, we remove all the invalid `session`(s) and deallocate al the associated `incarnation`(s).

NOTE: For dead incarnations that have not been closed we leave the files in the folder, since can't remove them from kernel space. Userspace will need to manually remove incarnations file for invalid processes. We consider as invalid a session which is associated to a dead process.

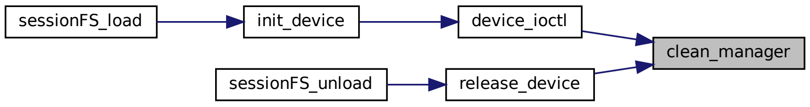
References delete_session(), delete_session_rcu(), MANAGER_EMPTY, remove_incarnation_info(), remove_session_info(), sessions, sessions_lock, and VALID_NODE.

Referenced by device_ioctl(), and release_device().

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.2.2 close_session()

```
int close_session (
            const char * pathname,
            int fdes,
            pid_t pid )
```

Closes a session.

Parameters

in	*pathname*	the pathname for the session containing the incarnation that is being closed.
in	*fdes*	The file descriptor of a session incarnation.
in	*pid*	The owner process pid.

Returns

0 on success or an error code.

This function will close one session, by finding the corresponding `incarnation`, using `search_session()`, copying the incarnation file over the original file (atomically in respect to other session operations on the same original file, and only if the `session` is valid), and deleting the incarnation, using `delete_incarnation()`. If after the incarnation deletion the `session` has no other `incarnation`(s) the it will also schedule the `session` to be removed. If we cannot locate the parent session of an incarnation `-EBADF` is returned.

To remove a session object we need to check several conditions:

- The `session` must be not in use by other threads (refcount==1)
- The `session` kernel object refcount, in the `info` member, must be 1
- The `session` must be still valid and not already marked for deletion

If the current `session` must be removed, we flag it as invalid, to avoid having new incarnations created in here before deallocating it and making sure that it will be eventually deallocated.

We also remove its information on SysFS using `remove_session_info()`.

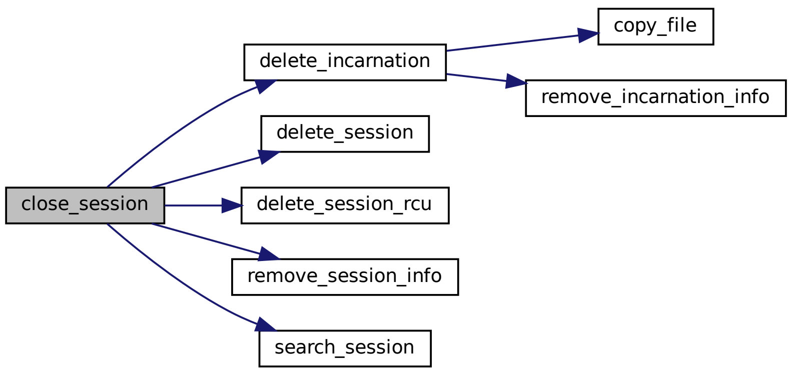
Then, we can remove the current `session` object from the rcu list, using the `sessions_lock` spinlock to avoid concurrent operations.

Finally, we try to deallocate the `session` if is invalid, using `delete_session()`.

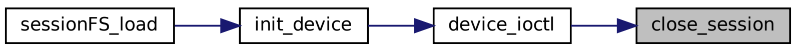
References delete_incarnation(), delete_session(), delete_session_rcu(), OVERWRITE_ORIG, remove_session_info(), search_session(), sessions_lock, and VALID_NODE.

Referenced by device_ioctl().

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.2.3 create_session()

```
struct incarnation* create_session (
            const char * pathname,
            int flags,
            pid_t pid,
            mode_t mode )
```

Create a new session for the specified file.

Parameters

in	*pathname*	The pathname of the file in which the session will be created.
in	*flags*	The flags that specify the permissions on the file.
in	*pid*	The pid of the process that wants to create the session.
in	*mode*	The permissions to apply to newly created files.

Returns

a pointer to an incarnation object, containing all the info on the current incarnation or an error code.

To create a new session we check if the original file was already opened with session semantic, by searching for an existing `session` with the same `pathname` using `search_session()`. If the found `session` is invalid or a matching `session` object is not found a new `session` object will be created, using `init_session()`. Then we create a new `incarnation` of the original file with `create_incarnation()`.

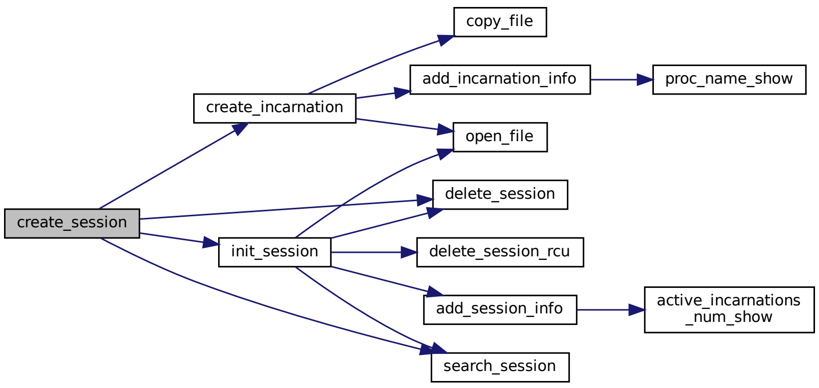
When the `incarnation` has been created the `refcount` of the parent session is decremented.

`-EAGAIN` is returned if the created session is invalid.

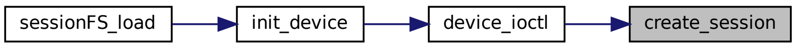
References create_incarnation(), delete_session(), init_session(), NO_FD, NO_PID, search_session(), and VALID_NODE.

Referenced by device_ioctl().

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.2.4 init_manager()

```
int init_manager (
            void  )
```

Initialization of the session manager data structures.

Returns

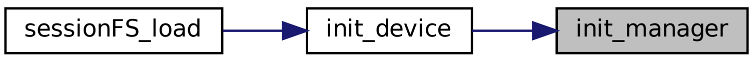
0 on success or an error code.

Initializes the `sessions` global variable as an empty list. Avoids the RCU initialization since we can't receive requests yet, so no one will use this list for now. Then initializes the `sessions_lock` spinlock.

References sessions, and sessions_lock.

Referenced by init_device().

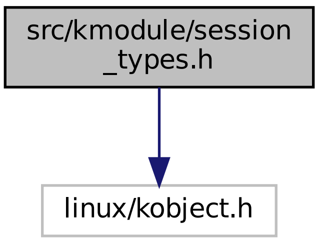
Here is the caller graph for this function:



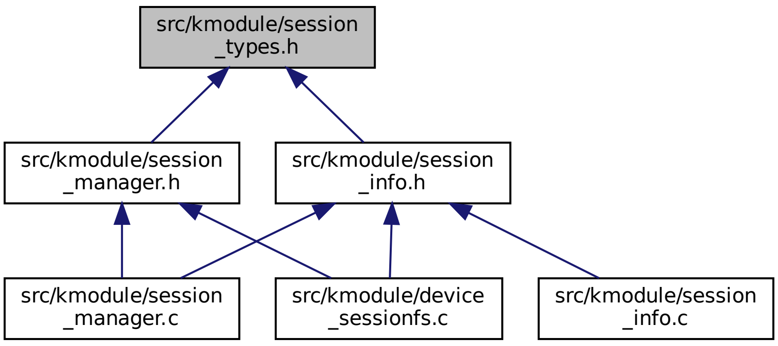
5.10 src/kmodule/session_types.h File Reference

```
#include <linux/kobject.h>
```

Include dependency graph for session_types.h:



This graph shows which files directly or indirectly include this file:


src/kmodule/session_types.h
src/kmodule/session_manager.h
src/kmodule/session_info.h
src/kmodule/session_manager.c
src/kmodule/device_sessionfs.c
src/kmodule/session_info.c

Data Structures

- struct incarnation

  Informations on an incarnation of a file.
- struct sess_info

  Infromations on a `session` used by SysFS.
- struct session

  General information on a `session`.
- struct session_rcu

  RCU item that contains a `session`.

5.10.1 Detailed Description

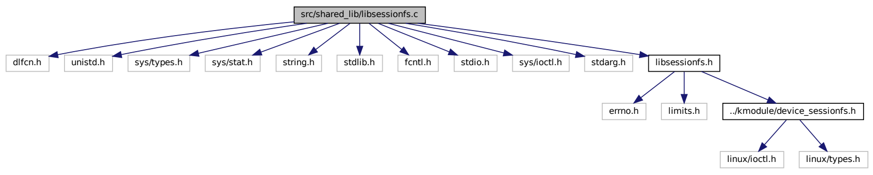
Struct definition common to all the module that need to manage session informations, component of the *Session Manager* submodule.

5.11 src/shared_lib/libsessionfs.c File Reference

Implementation of the userspace shared library.

```
#include <dlfcn.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <stdarg.h>
#include "libsessionfs.h"
```

Include dependency graph for libsessionfs.c:



Macros

- #define _GNU_SOURCE

  Enables RTLD_NEXT macro.
- #define DEV_PATH "/dev/SessionFS_dev"

  The path of our device file.

Typedefs

- typedef int(∗ orig_close_type) (int filedes)

  A typedef to aliases the function pointer to the libc `close`*.*
- typedef int(∗ orig_open_type) (const char ∗pathname, int flags)

  A typedef that aliases the function pointer to the libc `open`*.*

Functions

- static __attribute__ ((constructor))

  A program constructor which saves the original value for the `open` and `close` symbols.
- int close (int fd)

  Wraps the close determining if it must call the libc `close` or the SessionFS module.
- int device_shutdown (void)

  Asks to shut down the `SessionFS_dev` device.
- int get_sess_path (char ∗buf, int bufsize)

  Gets the current session path.
- int open (const char ∗pathname, int flags,...)

  Wraps the open determining if it must call the libc `open` or the SessionFS module.
- int write_sess_path (char ∗path)

  Changes the current session path.

Variables

- orig_close_type orig_close

  Global variable that holds the function pointer for libc `close`
- orig_open_type orig_open

  Global variable that holds the function pointer for libc `open`

5.11.1 Detailed Description

Implementation of the userspace shared library.

Used to provide a trasparent interface to the userspace application, that can use the libc open and close functions, along with the O_SESS flag to work with sessions. To change session path instead, it can use the get_sess_path() and write_sess_path() utility functions, to avoid the direct communication with the `SessionFS_dev` device.

5.11.2 Function Documentation

5.11.2.1 __attribute__()

```
static __attribute__ (
              (constructor) ) [static]
```

A program constructor which saves the original value for the `open` and `close` symbols.

Returns

0 on success or -1 on error, setting `errno`.

We save the original open and close since the library needs to understand when it's necessary to use the char device and when the libc implementation must be used. To do so we save them we use the `dlsym` function and we define two new types to avoid using a function pointer directly: `orig_open_type` and `orig_close_type`, these are simple typedefs that wrap the function poitner for libc `open` and `close`.

References orig_close, and orig_open.

5.11.2.2 close()

```
int close (
            int fd )
```

Wraps the close determining if it must call the libc `close` or the SessionFS module.

Parameters

in	*fd*	file descriptor to deallocate, same as libc `open`'s `fildes`.

Returns

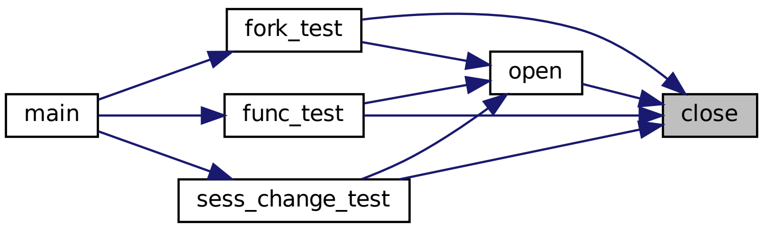
0 on success, -1 on error, setting `errno` to indicate the error value.

To determine if libc close must be used, the file pathname is read from `/proc/self/fd`, then `readlink` is used to resolve the pathname and make it absolute, finally, if this pathname contains the `_incarnation_[pid]_` substring then it must be closed by issuing an ioctl with number `IOCTL_SEQ_CLOSE` to the `SessionFS_dev` device. Otherwise the libc `close` is called. A sess_params struct is used to pass parameters to the char device when necessary. After the device completes its operations libc `close` is called to remove the file descriptor and `remove` is called to delete the incarnation file from the disk. If the return value from the ioctl is `-ENODEV` the the device was temporarly disabled and the operation must be retried.

References dev, DEV_PATH, IOCTL_SEQ_CLOSE, orig_close, orig_open, and sess_path.

Referenced by fork_test(), func_test(), open(), and sess_change_test().

Here is the caller graph for this function:



5.11.2.3 device_shutdown()

```
int device_shutdown (
            void )
```

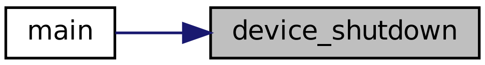
Asks to shut down the `SessionFS_dev` device.

To power down the device we only need to execute an ioctl with number `IOCTL_SEQ_SHUTDOWN` and the devce will proceed accordingly.

References dev, DEV_PATH, IOCTL_SEQ_SHUTDOWN, orig_close, and orig_open.

Referenced by main().

Here is the caller graph for this function:



5.11.2.4 get_sess_path()

```
int get_sess_path (
            char * buf,
            int bufsize )
```

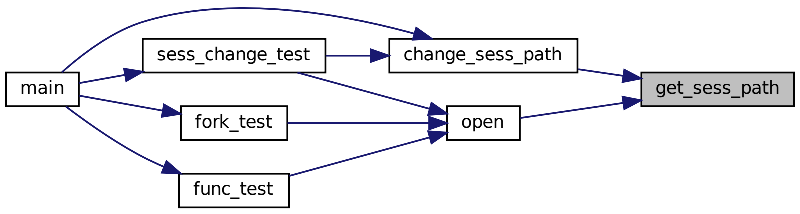
Gets the current session path.

This function is a simple utility function that reads from the `SessionFS_dev` device, located at `DEV_PATH`, the current session path and places it in the buffer provided by the caller.

References dev, DEV_PATH, orig_close, and orig_open.

Referenced by change_sess_path(), and open().

Here is the caller graph for this function:



5.11.2.5 open()

```
int open (
            const char * pathname,
            int flags,
             ... )
```

Wraps the open determining if it must call the libc `open` or the SessionFS module.

Parameters

in	*pathname*	The pathname of the file to be opened, same usage an type of the libc `open`'s `pathname`.
in	*flags*	Flags to determine the file status flag and the access modes, same as the libc `open`'s `oflag`, however a possible flag is the O_SESS flag which enables the session semantic.
in	*mode*	An optional parameter, which defines the permissions to set if a file must be created, (when `O_CREAT` is specified).

Returns

It will return a file descriptor if the operation is successful, or -1, setting `errno`.

This function will check for the presence of the O_SESS flag and if the path of the file to be opened has as a substring the session path.

If the checks are successful, the function will perform an ioctl call to the SessionFS kernel module, via the `SessionFS_dev` device, to open a new session for the given pathname. Otherwise, the function will call the libc implementation of the `open` systemcall.

To check that the given path has `sess_path` as a substring, `realpath()` is used, to convert the pathname to absolute. If `realpath()` fails with `ENOENT`, the path provided might be the relative path to a file that must be created, so the path of the current diretory is used as the file path.

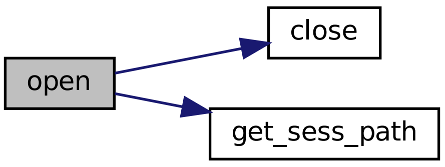
To perform the ioctl the `IOCTL_SEQ_OPEN` number is used and struct `sess_params` is filled and passed as an argument, to provide all the necessary informations to the device.

If the opened session is not valid, the function will call `close()` to remove the invalid session in a clean way and the function will fail with `EAGAIN`. When creating a new file we need to call `creat` since we have the symbol only for the open with two parametrs.

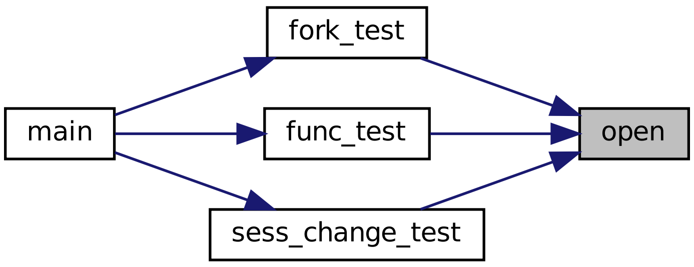
References close(), dev, DEV_PATH, get_sess_path(), IOCTL_SEQ_OPEN, O_SESS, orig_close, orig_open, sess_path, and VALID_SESS.

Referenced by fork_test(), func_test(), and sess_change_test().

Here is the call graph for this function:



Here is the caller graph for this function:



5.11.2.6 write_sess_path()

```
int write_sess_path (
             char * path )
```

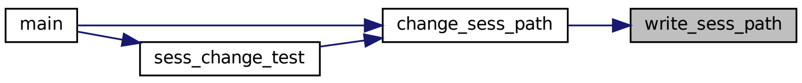
Changes the current session path.

This function is a simple utility function that writes on the `SessionFS_dev` device, located at `DEV_PATH`, the content of the buffer provided by the user; before doing so however, it uses the `realpath()` function to make sure that the path provided to char device is an absolute path.

References dev, DEV_PATH, orig_close, and orig_open.

Referenced by change_sess_path().

Here is the caller graph for this function:



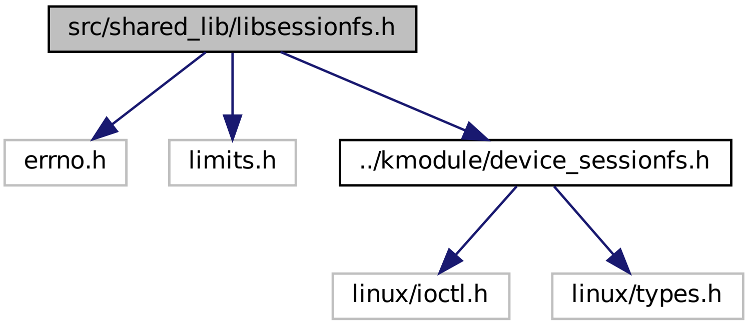
5.12 src/shared_lib/libsessionfs.h File Reference

Shared library header.

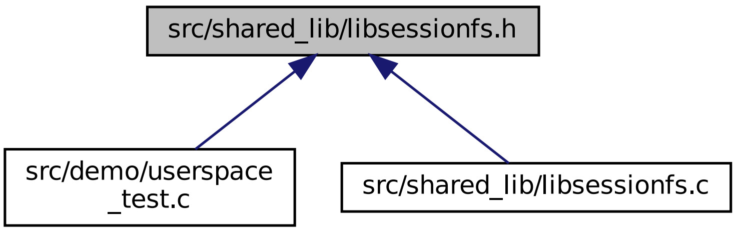
```
#include <errno.h>
#include <limits.h>
```

```
#include "../kmodule/device_sessionfs.h"
```

Include dependency graph for libsessionfs.h:



This graph shows which files directly or indirectly include this file:


src/shared_lib/libsessionfs.h
src/demo/userspace_test.c
src/shared_lib/libsessionfs.c

Functions

- int device_shutdown (void)

  Asks to shut down the `SessionFS_dev` device.
- int get_sess_path (char ∗buf, int bufsize)

  Gets the current session path.
- int write_sess_path (char ∗path)

  Changes the current session path.

5.12.1 Detailed Description

Shared library header.

Header file for the shared library that wraps the `open` and `close` functions. Contains only `get_sess_path()`, `write_sess_path()` and `device_shutdown()` since the `open()` and `close()` functions are used to wrap the libc syscalls and do not need to be exported.

5.12.2 Function Documentation

5.12.2.1 device_shutdown()

```
int device_shutdown (
             void  )
```

Asks to shut down the `SessionFS_dev` device.

Returns

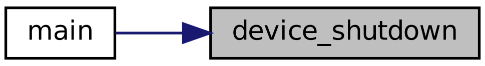
0 on success, `-EAGAIN` if the device is in use and cannot be removed.

To power down the device we only need to execute an ioctl with number `IOCTL_SEQ_SHUTDOWN` and the devce will proceed accordingly.

References dev, DEV_PATH, IOCTL_SEQ_SHUTDOWN, orig_close, and orig_open.

Referenced by main().

Here is the caller graph for this function:



5.12.2.2 get_sess_path()

```
int get_sess_path (
             char * buf,
             int bufsize )
```

Gets the current session path.

Parameters

out	*buf*	The buffer which will contain the output, must be provided.
in	*bufsize*	The length of the provided buffer.

Returns

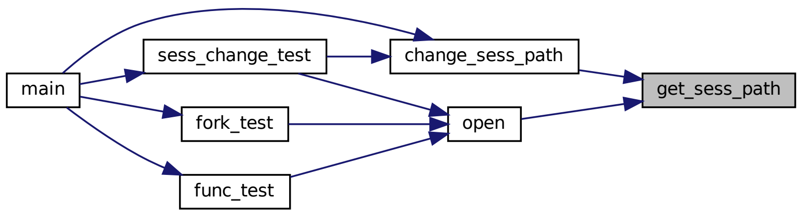
The number of bytes read or an error code.

This function is a simple utility function that reads from the `SessionFS_dev` device, located at `DEV_PATH`, the current session path and places it in the buffer provided by the caller.

References dev, DEV_PATH, orig_close, and orig_open.

Referenced by change_sess_path(), and open().

Here is the caller graph for this function:



5.12.2.3 write_sess_path()

```
int write_sess_path (
            char * path )
```

Changes the current session path.

Parameters

in	*buf*	The buffer which will contain the new path.

Returns

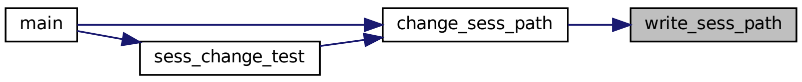
The number of bytes written or an error code.

This function is a simple utility function that writes on the `SessionFS_dev` device, located at `DEV_PATH`, the content of the buffer provided by the user; before doing so however, it uses the `realpath()` function to make sure that the path provided to char device is an absolute path.

References dev, DEV_PATH, orig_close, and orig_open.

Referenced by change_sess_path().

Here is the caller graph for this function:



Index