PennOS

1.0

Generated by Doxygen 1.10.0

Chapter 1

README

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 spthread_fwd_args_st Struct Reference

Data Fields

- pthread_fn **actual_routine**
- void * **actual_arg**
- bool **setup_done**
- pthread_mutex_t **setup_mutex**
- pthread_cond_t **setup_cond**
- spthread_meta_t * **child_meta**

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

- src/util/spthread.c

4.2 spthread_meta_st Struct Reference

Data Fields

- sigset_t **suspend_set**
- volatile sig_atomic_t **state**
- pthread_mutex_t **meta_mutex**

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

- src/util/spthread.c

4.3 spthread_signal_args_st Struct Reference

Data Fields

- const int **signal**
- volatile sig_atomic_t **ack**
- pthread_mutex_t **shutup_mutex**

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

- src/util/spthread.c

4.4 spthread_st Struct Reference

Data Fields

- pthread_t **thread**
- spthread_meta_t * **meta**

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

- src/util/spthread.h

Chapter 5

File Documentation

5.1 spthread.h

```
00001 #ifndef SPTHREAD_H_
00002 #define SPTHREAD_H_
00003
00004 #include <pthread.h>
00005 #include <stdbool.h>
00006
00007 // CAUTION: according to `man 7 pthread`:
00008 //
00009 //   On older Linux kernels, SIGUSR1 and SIGUSR2
00010 //   are used.  Applications must avoid the use of whichever set of
00011 //   signals is employed by the implementation.
00012 //
00013 // This may not work on other linux versions
00014
00015 // SIGNAL PTHREAD
00016 // NOTE: if within a created spthread you change
00017 // the behaviour of SIGUSR1, then you will not be able
00018 // to suspend and continue a spthread
00019 #define SIGPTHD SIGUSR1
00020
00021 // declares a struct, but the internals of the
00022 // struct cannot be seen by functions outside of spthread.c
00023 typedef struct spthread_meta_st spthread_meta_t;
00024
00025 // The spthread wrapper struct.
00026 // Sometimes you may have to access the inner pthread member
00027 // but you shouldn't need to do that
00028 typedef struct spthread_st {
00029   pthread_t thread;
00030   spthread_meta_t* meta;
00031 } spthread_t;
00032
00033 // NOTE:
00034 // None of these are signal safe
00035 // Also note that most of these functions are not safe to suspension,
00036 // meaning that if the thread calling these is an spthread and is suspended
00037 // in the middle of spthread_continue or spthread_suspend, then it may not work.
00038 //
00039 // Make sure that the calling thread cannot be suspended before calling these functions.
00040 // Exceptions to this are spthread_exit(), spthread_self() and if a thread is
00041 // continuing or suspending itself.
00042
00043 // spthread_create:
00044 // this function works similar to pthread_create, except for two differences.
00045 // 1) the created pthread is able to be asychronously suspended, and continued
00046 //    using the functions:
00047 //      - spthread_suspend
00048 //      - spthread_continue
00049 // 2) The created pthread will be suspended before it executes the specified
00050 //    routine. It must first be continued with `spthread_continue` before
00051 //    it will start executing.
00052 //
00053 // It is worth noting that this function is not signal safe.
00054 // In other words, it should not be called from a signal handler.
00055 //
00056 // to avoid repetition, see pthread_create(3) for details
00057 // on arguments and return values as they are the same here.
00058 int spthread_create(spthread_t* thread,
```

```
00059                     const pthread_attr_t* attr,
00060                     void* (*start_routine)(void*),
00061                     void* arg);
00062
00063 // The spthread_suspend function will signal to the
00064 // specified thread to suspend execution.
00065 //
00066 // Calling spthread_suspend on an already suspended
00067 // thread does not do anything.
00068 //
00069 // It is worth noting that this function is not signal safe.
00070 // In other words, it should not be called from a signal handler.
00071 //
00072 // args:
00073 // - pthread_t thread: the thread we want to suspend
00074 //   This thread must be created using the spthread_create() function,
00075 //   if created by some other function, the behaviour is undefined.
00076 //
00077 // returns:
00078 // - 0 on success
00079 // - EAGAIN if the thread could not be signaled
00080 // - ENOSYS if not supported on this system
00081 // - ESRCH if the thread specified is not a valid pthread
00082 int spthread_suspend(spthread_t thread);
00083
00084 // The spthread_suspend_self function will cause the calling
00085 // thread (which should be created by spthread_create) to suspend
00086 // itself.
00087 //
00088 // returns:
00089 // - 0 on success
00090 // - EAGAIN if the thread could not be signaled
00091 // - ENOSYS if not supported on this system
00092 // - ESRCH if the calling thread is not an spthread
00093 int spthread_suspend_self();
00094
00095 // The spthread_continue function will signal to the
00096 // specified thread to resume execution if suspended.
00097 //
00098 // Calling spthread_continue on an already non-suspended
00099 // thread does not do anything.
00100 //
00101 // It is worth noting that this function is not signal safe.
00102 // In other words, it should not be called from a signal handler.
00103 //
00104 // args:
00105 // - spthread_t thread: the thread we want to continue
00106 //   This thread must be created using the spthread_create() function,
00107 //   if created by some other function, the behaviour is undefined.
00108 //
00109 // returns:
00110 // - 0 on success
00111 // - EAGAIN if the thread could not be signaled
00112 // - ENOSYS if not supported on this system
00113 // - ESRCH if the thread specified is not a valid pthread
00114 int spthread_continue(spthread_t thread);
00115
00116 // The spthread_cancel function will send a
00117 // cancellation request to the specified thread.
00118 //
00119 // as of now, this function is identical to pthread_cancel(3)
00120 // so to avoid repitition, you should look there.
00121 //
00122 // Here are a few things that are worth highlighting:
00123 // - it is worth noting that it is a cancellation __request__
00124 //   the thread may not terminate immediately, instead the
00125 //   thread is checked whenever it calls a function that is
00126 //   marked as a cancellation point. At those points, it will
00127 //   start the cancellation procedure
00128 // - to make sure all things are de-allocated properly on
00129 //   normal exiting of the thread and when it is cancelled,
00130 //   you should mark a defered de-allocation with
00131 //   pthread_cleanup_push(3).
00132 //   consider the following example:
00133 //
00134 //     void* thread_routine(void* arg) {
00135 //        int* num = malloc(sizeof(int));
00136 //        pthread_cleanup_push(&free, num);
00137 //        return NULL;
00138 //     }
00139 //
00140 //    this program will allocate an integer on the heap
00141 //    and mark that data to be de-allocated on cleanup.
00142 //    This means that when the thread returns from the
00143 //    routine specified in spthread_create, free will
00144 //    be called on num. This will also happen if the thread
00145 //    is cancelled and not able to be exited normally.
```

```
00146 //
00147 //    Another function that should be used in conjunction
00148 //    is pthread_cleanup_pop(3). I will leave that
00149 //    to you to read more on.
00150 //
00151 // It is worth noting that this function is not signal safe.
00152 // In other words, it should not be called from a signal handler.
00153 //
00154 // args:
00155 // - spthread_t thread: the thread we want to cancel.
00156 //   This thread must be created using the spthread_create() function,
00157 //   if created by some other function, the behaviour is undefined.
00158 //
00159 // returns:
00160 // - 0 on success
00161 // - ESRCH if the thread specified is not a valid pthread
00162 int spthread_cancel(spthread_t thread);
00163
00164 // Can be called by a thread to get two peices of information:
00165 // 1. Whether or not the calling thread is an spthread (true or false)
00166 // 2. The spthread_t of the calling thread, if it is an spthread_t
00167 //
00168 // almost always the function will be called like this:
00169 // spthread_t self;
00170 // bool i_am_spthread = spthread_self(&self);
00171 //
00172 // args:
00173 // - spthread_t* thread: the output parameter to get the spthread_t
00174 //   representing the calling thread, if it is an spthread
00175 //
00176 // returns:
00177 // - true if the calling thread is an spthread_t
00178 // - false otherwise.
00179 bool spthread_self(spthread_t* thread);
00180
00181 // The equivalent of pthread_join but for spthread
00182 // To make sure all resources are cleaned up appropriately
00183 // spthreads that are created must at some ppoint have spthread_join
00184 // called on them. Do not use pthread_join on an spthread.
00185 //
00186 // to avoid repetition, see pthread_join(3) for details
00187 // on arguments and return values as they are the same as this function.
00188 int spthread_join(spthread_t thread, void** retval);
00189
00190 // The equivalent of pthread_exit but for spthread
00191 // spthread_exit must be used by spthreads instead of pthread_exit.
00192 // Otherwise, calls to spthread_join or other functions (like spthread_suspend)
00193 // may not work as intended.
00194 //
00195 // to avoid repetition, see pthread_exit(3) for details
00196 // on arguments and return values as they are the same as this function.
00197 void spthread_exit(void* status);
00198
00199 #endif  // SPTHREAD_H_
```

Index