PiP - Process-in-Process

Generated by Doxygen 1.8.5

Sat Jul 4 2020 17:26:59

Contents

Chapter 1

Proces-in-Process (PiP) Overview

Process-in-Process (PiP)

PiP is a user-level library to have the best of the both worlds of multi-process and multi-thread parallel execution models. PiP allows a process to create sub-processes into the same virtual address space where the parent process runs. The parent process and sub-processes share the same address space, however, each process has its own variable set. So, each process runs independently from the other process. If some or all processes agree, then data own by a process can be accessed by the other processes. Those processes share the same address space, just like pthreads, and each process has its own variables like a process. The parent process is called PiP process and a sub-process are called a PiP task.

PiP Versions

Currently there are three PiP library versions:

- Version 1 - Deprecated
- Version 2 - Stable version
- Version 3 - Stable version supporting BLT and ULP

In this document, **N** denotes the PiP version number.

Bi-Level Thread (BLT, from v3)

PiP also provides new thread implementation named "Bi-Level Thread (BLT)", again, to take the best of two worlds, Kernel-Level Thread (KLT) and User-Level Thread (ULT) here. A BLT is a PiP task. When a PiP task is created it runs as a KLT. At any point the KLT can becomme a ULT by decoupling the associated kernel thread from the KLT. The decoupled kernel thread becommes idle. Later, the ULT can become KLT again by coupling with the kernel thread.

User-Level Process (ULP, from v3)

As described, PiP allows PiP tasks to share the same virtual address space. This mans that a PiP task can context-switch to the other PiP task at user-level. This is called User-Level Process where processes may be derived from the same program or different programs. Threads basically share most of the kernel resources, such as address space, file descriptors, a process id, and so on whilst processes do not. Every process has its ows file descriptor space, for example. When a ULP is scheduled by a KLT having PID 1000, then the getpid() is called by the U-LP returns 1000. Further, when the ULT is migrated to be scheduled by the other KLT, then the returned PID is different. So, when implemnting a ULP system, this systemcall consistency must be preserved. In ULP on PiP, the

consistency can be maintained by utilizing the above BLT mechanism. When a ULT tries to call a system call, it is coupled with its kernel thread which was created at the beginning as a KLT. It should be note that Thread Local Storage (TLS) regions are also switched when switching ULP (and BLT) contexts.

Execution Mode

There are several PiP implementation modes which can be selected at the runtime. These implementations can be categorized into two according to the behavior of PiP tasks,

- Process and
- (P)Thread

In the pthread mode, although each PiP task has its own variables unlike thread, PiP task behaves more like P-Thread, having a TID, having the same file descriptor space, having the same signal delivery semantics as Pthread does, and so on. In the process mode, PiP task behaves more like a process, having a PID, having an independent file descriptor space, having the same signal delivery semantics as Linux process does, and so on. The above mentioned ULP can only work with the process mode.

When the `PIP_MODE` environment variable set to "thread" or "pthread" then the PiP library runs based on the pthread mode, and it is set to "process" then it runs with the process mode. There are also three implementations in the process mode; "process:preload," "process:pipclone" and "process:got." The "process:preload" mode must be with the LD_PRELOAD environment variable setting so that the clone() system call wrapper can work with. The "process:pipclone" mode can only be specified with the PIP-patched glibc library (see below: GLIBC issues).

There several function provided by the PiP library to absorb the difference due to the execution mode

License

This project is licensed under the 2-clause simplified BSD License - see the LICENSE file for details.

Installation

PiP Trial by using Docker image

Download and run the PiP Docker image.

```
$ docker pull rikenpip/pip-vN
$ sudo docker run -it rikenpip/pip-vN /bin/bash
```

Source Repositories

The installation of PiP related packages must follow the order below;

1. Build PiP-glibc (optional)
2. Build PiP
3. Build PiP-gdb (optional)

Note that if PiP-gdb will not work at all without PiP-glibc. Further, PiP can only create up to around ten PiP tasks without installing PiP-glibc.

- `PiP-glibc` - patched GNU libc for PiP

- PiP - Process in Process (this package)
- PiP-gdb - patched gdb to debug PiP root and PiP tasks.

Before installing PiP, we strongly recommend you to install PiP-glibc.

After installing PiP, PiP-gdb can be installed too.

Installation from the source code.

1. Building PiP-glibc (optional)

   Fetch source tree (CentOS7 or RHEL7):

   $ git clone -b pip-centos7 git@git.sys.aics.riken.jp:software/PIP-glibc

   Fetch source tree (CentOS8 or RHEL8):

   $ git clone -b pip-centos8 git@git.sys.aics.riken.jp:software/PIP-glibc

   Build PiP-glibc

   $ mkdir GLIBC_BUILD_DIR $ cd GLIBC_BUILD_DIR $ GLIBC_SRC_DIR/build.sh –prefix=GLIBC_INSTAL-L_DIR

2. Build PiP library

   The same source code can be ussed for CentOS7 and CentOS8 (RHEL7 and RHEL8).

   $ git clone -b pip-N git@git.sys.aics.riken.jp:software/PiP $ cd PIP_SRC_DIR $ ./configure –prefix=PIP_INSTALL_DIR [–with-glibc-libdir=GLIBC_INSTALL_DIR/lib] $ make install doxgyen-install $ cd PIP_INSTALL_DIR/bin $ ./piplnlibs

   If you want to make sure if the PiP library is correctly installed, then do the following;

   $ cd PIP_SRC_DIR $ make install-test

   Important note: The prefix directory of PiP-glibc and the prefix directory of PiP itself must NOT be the same.

3. Build PiP-gdb (optional)

   Fetch source tree (CentOS7 or RHEL7):

   $ git clone -b pip-centos7 git@git.sys.aics.riken.jp:software/PIP-gdb

   Ftech source tree (CentOS8 or RHEL8):

   $ git clone -b pip-centos8 git@git.sys.aics.riken.jp:software/PIP-gdb

   Build PiP-gdb

   $ cd GLIBC_SRC_DIR $ ./build.sh –prefix=GLIBC_INSTALL_DIR –with-pip=PIP_INSTALL_DIR

   The prefix directory of PiP-gdb can be the same with the prefix directory of PiP library.

Installation from RPMs

RPM packages and their yum repository are also available for CentOS 7 / RHEL7.

```
$ sudo rpm -Uvh https://git.sys.r-ccs.riken.jp/PiP/package/el/7/noarch/pip-1/pip-release-N-0.noarch.rpm
$ sudo yum install pip-glibc
$ sudo yum install pip pip-debuginfo
$ sudo yum install pip-gdb
```

If PiP packages are installed by the above RPMs, **PIP_INSTALL_DIR** is "/usr."

PiP documents

The following PiP documents are created by using Doxygen.

Man pages

Man pages will be installed at **PIP_INSTALL_DIR**/share/man.

```
$ man -M PIP_INSTALL_DIR/share/man 7 libpip
```

Or, use the pip-man command (from v2).

```
$ PIP_INSTALL_DIR/bin/pip-man 7 libpip
```

The above two exammples will show you the same document you are reading.

PDF

PDF documents will be installed at **PIP_INSTALL_DIR**/share/doc/pip/pdf.

Getting Started

Compile and link your PiP programs

- pipcc(1) command (since v2)

You can use pipcc(1) command to compile and link your PiP programs.

```
$ pipcc -Wall -O2 -g -c pip-prog.c
$ pipcc -Wall -O2 -g -o pip-prog pip-prog.c
```

Run your PiP programs

- pip-exec(1) command (piprun(1) in PiP v1)

Let's assume your that have a non-PiP program(s) and wnat to run as PiP tasks. All you have to do is to compile your program by using the above pipcc(1) command and to use the pip-exec(1) command to run your program as PiP tasks.

```
$ pipcc myprog.c -o myprog
$ pip-exec -n 8 ./myprog
$ ./myprog
```

In this case, the pip-exec(1) command becomes the PiP root and your program runs as 8 PiP tasks. Your program can also run as a normal (non-PiP) program without using the pip-exec(1) command. Note that the 'myprog.c' may or may not call any PiP functions.

You may write your own PiP programs whcih includes the PiP root programming. In this case, your program can run without using the pip-exec(1) command.

If you get the following message when you try to run your program;

```
PiP-ERR(19673) './myprog' is not PIE
```

Then this means that the 'myprog' is not compiled by using the pipcc(1) command properly. You may check if your program(s) can run as a PiP root and/or PiP task by using the pip-check(1) command (from v2);

```
$ pip-check a.out
a.out : Root&Task
```

Above example shows that the 'a.out' program can run as a PiP root and PiP tasks.

- pips(1) command (from v2)

  You can check if your PiP program is running or not by using the pips(1) command.

List the PiP tasks via the 'ps' command;

```
$ pips -l [ COMMAND ]
```

or, show the activities of PiP tasks via the 'top' command;

```
$ pips -t [ COMMAND ]
```

Here **COMMAND** is the name (not a path) of PiP program you are running.

Additionally you can kill all of your PiP tasks by using the same pips(1) command;

```
$ pips -s KILL [ COMMAND ]
```

Debugging your PiP programs by the pip-gdb command

The following procedure attaches all PiP tasks, which are created by same PiP root task, as GDB inferiors.

```
$ pip-gdb
(gdb) attach PID
```

The attached inferiors can be seen by the following GDB command:

```
(gdb) info inferiors
  Num  Description              Executable
  4    process 6453 (pip 2)     /somewhere/pip-task-2
  3    process 6452 (pip 1)     /somewhere/pip-task-1
  2    process 6451 (pip 0)     /somewhere/pip-task-0
* 1    process 6450 (pip root)  /somewhere/pip-root
```

You can select and debug an inferior by the following GDB command:

```
(gdb) inferior 2
[Switching to inferior 2 [process 6451 (pip 0)] (/somewhere/pip-task-0)]
```

When an already-attached program calls 'pip_spawn()' and becomes a PiP root task, the newly created PiP child tasks aren't attached automatically, but you can add empty inferiors and then attach the PiP child tasks to the inferiors. e.g.

```
.... type Control-Z to stop the root task.
^Z
Program received signal SIGTSTP, Stopped (user).

(gdb) add-inferior
Added inferior 2
(gdb) inferior 2
(gdb) attach 1902

(gdb) add-inferior
Added inferior 3
(gdb) inferior 3
(gdb) attach 1903

(gdb) add-inferior
Added inferior 4
(gdb) inferior 4
(gdb) attach 1904

(gdb) info inferiors
  Num  Description              Executable
* 4    process 1904 (pip 2)     /somewhere/pip-task-2
  3    process 1903 (pip 1)     /somewhere/pip-task-1
  2    process 1902 (pip 0)     /somewhere/pip-task-0
  1    process 1897 (pip root)  /somewhere/pip-root
```

You can attach all relevant PiP tasks by:

```
$ pip-gdb -p PID-of-your-PiP-program
```

(from v2)

If the PIP_GDB_PATH environment is set to the path pointing to PiP-gdb executable file, then PiP-gdb is automatically attached when an excetion signal (SIGSEGV and SIGHUP by default) is delivered. The exception signals can also be defined by setting the PIP_GDB_SIGNALS environment. Signal names (case insensitive) can be concatenated by the '+' or '-' symbol. 'all' is reserved to specify most of the signals. For example, 'ALL-TERM' means all signals excepting SIGTERM, another example, 'PIPE+INT' means SIGPIPE and SIGINT. If one of the defined or default signals is delivered, then PiP-gdb will be attached. The PiP-gdb will show backtrace by default. If users specify PIP_GDB_COMMAND that a filename containing some GDB commands, then those GDB commands will be executed by the GDB, instead of backtrace, in batch mode. If the PIP_STOP_ON_START environment is set (to any value), then the PiP library delivers SIGSTOP to a spawned PiP task which is about to start user program.

FAQ

- Does MPI with PiP exist? Currently, we are working with ANL to develop MPICH using PiP. This repository, located at ANL, is not yet open to public at the time of this writing.

Publications

Research papers

A. Hori, M. Si, B. Gerofi, M. Takagi, J. Dayal, P. Balaji, and Y. Ishikawa. "Process-in-process: techniques for practical address-space sharing," In Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '18). ACM, New York, NY, USA, 131-143. DOI: https://doi.org/10.1145/3208040.3208045

Presentation Slides

- HPDC'18
- ROSS'18
- IPDPS/RADR'20

Mailing List

pip@ml.riken.jp

PiP Man Pages

Overview

Commands

- pipcc
- piplnlibs

- pips
- printpipmode

Functions

- pip_abort
- pip_barrier_fin
- pip_barrier_init
- pip_barrier_wait
- pip_blt_spawn
- pip_couple
- pip_decouple
- pip_dequeue_and_resume
- pip_dequeue_and_resume_N
- pip_dequeue_and_resume_N_nolock
- pip_dequeue_and_resume_nolock
- pip_exit
- pip_export
- pip_fin
- pip_get_aux
- pip_get_mode
- pip_get_mode_str
- pip_get_ntasks
- pip_get_pipid
- pip_get_sched_domain
- pip_get_system_id
- pip_get_task_by_pipid
- pip_get_task_pipid
- pip_import
- pip_init
- pip_isa_root
- pip_isa_task
- pip_is_initialized
- pip_is_shared_fd
- pip_is_threaded
- pip_kill
- pip_kill_all_tasks

- pip_mutex_fin
- pip_mutex_init
- pip_mutex_lock
- pip_mutex_unlock
- pip_named_export
- pip_named_import
- pip_named_tryimport
- pip_set_aux
- pip_sigmask
- pip_signal_wait
- pip_spawn
- pip_spawn_from_func
- pip_spawn_from_main
- pip_spawn_hook
- pip_suspend_and_enqueue
- pip_suspend_and_enqueue_nolock
- pip_task_queue_count
- pip_task_queue_dequeue
- pip_task_queue_describe
- pip_task_queue_enqueue
- pip_task_queue_fin
- pip_task_queue_init
- pip_task_queue_isempty
- pip_task_queue_lock
- pip_task_queue_trylock
- pip_task_queue_unlock
- pip_task_self
- pip_task_spawn
- pip_trywait
- pip_trywait_any
- pip_wait
- pip_wait_any
- pip_yield
- pip_yield_to

Author

Atsushi Hori

Riken Center for Commputational Science (R-CCS)

Japan

Chapter 2

PiP Commands

2.1 pipcc

C compiler driver for PiP

Synopsis

pipcc [PIP-OPTIONS] [CC-COMMAND-OPTIONS_AND_ARGS]

Parameters

-piproot	the compile (and link) as a PiP root
-piptask	the compile (and link) as a PiP task
-nopip	No PiP related settings will be applied

Note

The **-piproot** and **-piptask** options can be specified at the same time. In this case, the compiled object can be both of PiP root and PiP task. This is also the default behavior when none of them is not specified.

Environment

if `CC` environment is set then `$`(CC) will be used as a C compiler

See Also

pip-exec(1), pip-mode(1)

2.2 pip-check

PiP binary checking program if a progarm can run sa a PiP root and/or PiP task

Synopsis

pipcheck [OPTION] pip-prog [...]

Parameters

-r	check if a.out can be PiP root
-t	check if a.out can be PiP task
-b	check if a.out can be PiP root and/or PiP task
-v	show reason
-h	show this message

See Also

pipcc(1)

2.3 pip-exec

run program(s) as PiP tasks

Synopsis

pip-exec [OPTIONS] <program> ... [: ...]

Description

Run a program as PiP task(s). Mutiple programs can be specified by separating them with ':' to share the same virtual address space with the `pip-exec` command.

Parameters

-n N	number of tasks
-f FUNC	function name to start
-c CORE	specify the CPU core number to bind core(s)
-r	core binding in the round-robin fashion

See Also

pipcc(1)

2.4 piplnlibs

command to create symbolic links to the SOLIBs in the install directory of the patched GLIBC.

Synopsis

piplnlibs [OPTIONS]

Description

This command creates a number of symbolic links to the SOLIBs which are not installed by the patched GLIBC installation.

Parameters

-r	Remove symbolic links to SOLIBs in /home/ahori/PiP/x86_64/install/lib before creating.

-s	Silent mode.

2.5 pip-man

show PiP man page

Synopsis

pip-man [MAN-OPT] MAN-TOPIC

Description

Show PiP man pages. It can also accept the man command options.

See Also

man(1)

2.6 pip-mode

Set PiP execution mode

Synopsis

pip-mode [OPTION] [PIP-COMMAND]

Description

The following options are avilable. If no of them specified, then the compiled output file can be used as both PiP root and PiP task.

Parameters

-P	'process' mode
-L	'process:preload' mode
-C	'process:clone' mode
-G	'process:got' mode
-T	'thread' mode
-u	Show usage

See Also

pip-exec(1)

2.7 pips

List or kill running PiP tasks

Synopsis

pips [OPTION] [PIP-PROG-NAME ...]

Description

The following options are avilable.

Parameters

-s SIGNAL	Send the specified signal to the specified PiP tasks
-k	Same as 'pips -s TERM'
-l	List (ps command) running PiP tasks specified. This is the default action.
--list	Same as 'pips -l'
-t	Show running PiP tasks specified by using the top command.
-v	Verbose mode
--top	Same sa 'pip -t'

See Also

ps(1), top(1)

2.8 printpipmode

Print current PiP mode

Synopsis

printpipmode

See Also

pip-mode(1)

Chapter 3

PiP Functions

3.1 PiP Initialization/Finalization

Functions

- int **pip_init** (int *pipidp, int *ntasks, void **root_expp, uint32_t opts)
- int **pip_fin** (void)

3.1.1 Detailed Description

3.1.1.1 PiP Initialization/Finalization

Description

PiP initialization/finalization functions

3.1.1.2 pip_init

Name

Initialize the PiP library

Synopsis

#include <pip.h>
int pip_init(int *pipidp, int *ntasks, void **root_expp, uint32_t opts);

Description

This function initializes the PiP library. The PiP root process must call this. A PiP task is not required to call this function unless the PiP task calls any PiP functions.

When this function is called by a PiP root, `ntasks`, and `root_expp` are input parameters. If this is called by a PiP task, then those parameters are output returning the same values input by the root.

A PiP task may or may not call this function. If `pip_init` is not called by a PiP task explicitly, then `pip_init` is called magically and implicitly even if the PiP task program is NOT linked with the PiP library.

Parameters

out	*pipidp*	When this is called by the PiP root process, then this returns PIP_PIPID_ROOT, otherwise it returns the PiP ID of the calling PiP task.
in,out	*ntasks*	When called by the PiP root, it specifies the maximum number of PiP tasks. When called by a PiP task, then it returns the number specified by the PiP root.
in,out	*root_expp*	If the root PiP is ready to export a memory region to any PiP task(s), then this parameter is to pass the exporting address. If the PiP root is not ready to export or has nothing to export then this variable can be NULL. When called by a PiP task, it returns the exported address by the PiP root, if any.
in	*opts*	Specifying the PiP execution mode and See below.

Notes

The opts may have one of the defined values PIP_MODE_PTHREAD, PIP_MODE_PROCESS, PIP_MODE_PROCESS_PRELOAD, PIP_MODE_PROCESS_PIPCLONE and PIP_MODE_PROCESS_GOT, or any combination (bit-wise or) of them. If combined or opts is zero, then an appropriate one is chosen by the library. This PiP execution mode can be specified by an environment variable described below.

Returns

Zero is returned if this function succeeds. Otherwise an error number is returned.

Return values

EINVAL	*ntasks* is negative
EBUSY	PiP root called this function twice or more without calling pip_fin(1).
EPERM	*opts* is invalid or unacceptable
EOVERFLOW	*ntasks* is too large
ELIBSCN	verssion miss-match between PiP root and PiP task

Environment

- **PIP_MODE** Specifying the PiP execution mmode. Its value can be either thread, pthread, process, process:preload, process:pipclone, or process:got.
- **LD_PRELOAD** This is required to set appropriately to hold the path to pip_preload.so file, if the PiP execution mode is PIP_MODE_PROCESS_PRELOAD (the opts in pip_init) and/or the PIP_MODE ennvironment is set to process:preload. See also the pip_mode(1) command to set the environment variable appropriately and easily.
- **PIP_GDB_PATH** If thisenvironment is set to the path pointing to the PiP-gdb executable file, then PiP-gdb is automatically attached when an excetion signal (SIGSEGV and SIGHUP by default) is delivered. The signals which triggers the PiP-gdb invokation can be specified the PIP_GDB_SIGNALS environment described below.
- **PIP_GDB_COMMAND** If this PIP_GDB_COMMAND is set to a filename containing some GDB commands, then those GDB commands will be executed by the GDB in batch mode, instead of backtrace.
- **PIP_GDB_SIGNALS** Specifying the signal(s) resulting automatic PiP-gdb attach. Signal names (case insensitive) can be concatenated by the '+' or '-' symbol. 'all' is reserved to specify most of the signals. For example, 'ALL-TERM' means all signals excepting SIGTERM, another example, 'PIPE+INT' means SIGPIPE and SIGINT. Some signals such as SIGKILL and SIGCONT cannot be specified.
- **PIP_SHOW_MAPS** If the value is 'on' and one of the above exection signals is delivered, then the memory map will be shown.
- **PIP_SHOW_PIPS** If the value is 'on' and one of the above exection signals is delivered, then the process status by using the pips command (see also pips(1)) will be shown.

Bugs

Is is NOT guaranteed that users can spawn tasks up to the number specified by the *ntasks* argument. There are some limitations come from outside of the PiP library (from GLIBC).

See Also

pip_named_export(3), pip_export(3), pip_fin(3), pip-mode(1), pips(1)

3.1.1.3 pip_fin

Name

Finalize the PiP library

Synopsis

#include <pip.h>
int pip_fin(void);

Description

This function finalizes the PiP library. After calling this, most of the PiP functions will return the error code `EPERM`.

Returns

zero is returned if this function succeeds. On error, error number is returned.

Return values

EPERM	`pip_init` is not yet called
EBUSY	`one` or more PiP tasks are not yet terminated

Notes

The behavior of calling `pip_init` after calling this `pip_fin` is note defined and recommended to do so.

See Also

pip_init(3)

3.2 Spawning PiP task

Functions

- void **pip_spawn_from_main** (pip_spawn_program_t *progp, char *prog, char **argv, char **envv, void *exp)
- void **pip_spawn_from_func** (pip_spawn_program_t *progp, char *prog, char *funcname, void *arg, char **envv, void *exp)
- void **pip_spawn_hook** (pip_spawn_hook_t *hook, pip_spawnhook_t before, pip_spawnhook_t after, void *hookarg)
- int **pip_task_spawn** (pip_spawn_program_t *progp, uint32_t coreno, uint32_t opts, int *pipidp, pip_spawn_hook_t *hookp)
- int **pip_spawn** (char *filename, char **argv, char **envv, uint32_t coreno, int *pipidp, pip_spawnhook_t before, pip_spawnhook_t after, void *hookarg)
- int **pip_blt_spawn** (pip_spawn_program_t *progp, uint32_t coreno, uint32_t opts, int *pipidp, pip_task_t **bltp, pip_task_queue_t *queue, pip_spawn_hook_t *hookp)

3.2.1 Detailed Description

3.2.1.1 PiP Spawnig PiP (ULP/BLT) task

Description

Spawning PiP task or ULP/BLT task

3.2.1.2 pip_spawn_from_main

Setting information to invoke a PiP task starting from the main function

Synopsis

#include <pip.h>
void pip_spawn_from_main(pip_spawn_program_t *progp, char *prog, char **argv, char **envv, void *exp)

Description

This function sets up the `pip_spawn_program_t` structure for spawning a PiP task, starting from the mmain function.

Parameters

`out`	*progp*	Pointer to the `pip_spawn_program_t` structure in which the program invokation information will be set
`in`	*prog*	Path to the executiable file.
`in`	*argv*	Argument vector.
`in`	*envv*	Environment variables. If this is `NULL`, then the `environ` variable is used for the spawning PiP task.
`in`	*exp*	Export value to the spawning PiP task

See Also

pip_task_spawn(3), pip_spawn_from_func(3)

3.2.1.3 pip_spawn_from_func

Setting information to invoke a PiP task starting from a function defined in a program

Synopsis

#include <pip.h>
pip_spawn_from_func(pip_spawn_program_t *progp, char *prog, char *funcname, void *arg, char **envv, void *exp);

Description

This function sets the required information to invoke a program, starting from the `main()` function. The function should have the function prototype as shown below;

```
int start_func( void *arg )
```

This start function must be globally defined in the program.. The returned integer of the start function will be treated in the same way as the `main` function. This implies that the `pip_wait` function family called from the PiP root can retrieve the return code.

Parameters

out	*progp*	Pointer to the `pip_spawn_program_t` structure in which the program invokation information will be set
in	*prog*	Path to the executiable file.
in	*funcname*	Function name to be started
in	*arg*	Argument which will be passed to the start function
in	*envv*	Environment variables. If this is `NULL`, then the `environ` variable is used for the spawning PiP task.
in	*exp*	Export value to the spawning PiP task

See Also

pip_task_spawn(3), pip_spawn_from_main(3)

3.2.1.4 pip_spawn_hook

Setting invocation hook information

Synopsis

#include <pip.h>
void pip_spawn_hook(pip_spawn_hook_t *hook, pip_spawnhook_t before, pip_spawnhook_t after, void *hookarg);

Description

The *before* and *after* functions are introduced to follow the programming model of the `fork` and `exec`. *before* function does the prologue found between the `fork` and `exec`. *after* function is to free the argument if it is `malloc()ed`, for example.

Precondition

It should be noted that the *before* and *after* functions are called in the *context* of PiP root, although they are running as a part of PiP task (i.e., having PID of the spawning PiP task). Conversely speaking, those functions cannot access the variables defined in the spawning PiP task.
The before and after hook functions should have the function prototype as shown below;

```
int hook_func( void *hookarg )
```

Parameters

out	*hook*	Pointer to the `pip_spawn_hook_t` structure in which the invocation hook information will be set
in	*before*	Just before the executing of the spawned PiP task, this function is called so that file descriptors inherited from the PiP root, for example, can deal with. This is only effective with the PiP process mode. This function is called with the argument *hookarg* described below.
in	*after*	This function is called when the PiP task terminates for the cleanup purpose. This function is called with the argument *hookarg* described below.
in	*hookarg*	The argument for the *before* and *after* function call.

Note

Note that the file descriptors and signal handlers are shared between PiP root and PiP tasks in the pthread execution mode.

See Also

pip_task_spawn(3)

3.2.1.5 pip_task_spawn

Spawning a PiP task

Synopsis

#include <pip.h>
int pip_task_spawn(pip_spawn_program_t *progp, uint32_t coreno, uint32_t opts, int *pipidp, pip_spawn_hook_t *hookp);

Description

This function spawns a PiP task specified by `progp`.

In the process execution mode, the file descriptors having the `FD_CLOEXEC` flag is closed and will not be passed to the spawned PiP task. This simulated close-on-exec will not take place in the pthread execution mode.

Parameters

`out`	*hook*	Pointer to the `pip_spawn_hook_t` structure in which the invocation hook information is set
`in`	*coreno*	CPU core number for the PiP task to be bound to. By default, `coreno` is set to zero, for example, then the calling task will be bound to the first core available. This is in mind that the available core numbers are not contiguous. To specify an absolute core number, `coreno` must be bitwise-ORed with `PIP_CPUCORE_ABS`. If `PIP_CPUCORE_ASIS` is specified, then the core binding will not take place.
`in`	*opts*	option flags
`in,out`	*pipidp*	Specify PiP ID of the spawned PiP task. If `PIP_PIPID_ANY` is specified, then the PiP ID of the spawned PiP task is up to the PiP library and the assigned PiP ID will be returned.
`in`	*hookp*	Hook information to be invoked before and after the program invokation.

Returns

Zero is returned if this function succeeds. On error, an error number is returned.

Return values

EPERM	PiP library is not yet initialized
EPERM	PiP task tries to spawn child task
EINVAL	`progp` is `NULL`
EINVAL	`opts` is invalid and/or unacceptable
EINVAL	the value off `pipidp` is invalid
EINVAL	the coreno is larger than or equal to `PIP_CPUCORE_CORENO_MAX`
EBUSY	specified PiP ID is alredy occupied
ENOMEM	not enough memory
ENXIO	`dlmopen` failss

Note

In the process execution mode, each PiP task may have its own file descriptors, signal handlers, and so on, just like a process. Contrastingly, in the pthread executionn mode, file descriptors and signal handlers are shared among PiP root and PiP tasks while maintaining the privatized variables.

Environment

- **PIP_STOP_ON_START** Specifying the PIP ID to stop on start to debug the specified PiP task from the beginning. If the before hook is specified, then the PiP task will be stopped just before calling the before hook.

Bugs

In theory, there is no reason to restrict for a PiP task to spawn another PiP task. However, the current glibc implementation does not allow to do so.

If the root process is multithreaded, only the main thread can call this function.

See Also

pip_task_spawn(3), pip_spawn_from_main(3), pip_spawn_from_func(3), pip_spawn_hook(3), pip_spawn(3), pip_blt_spawn(3)

3.2.1.6 pip_spawn

spawn a PiP task (PiP v1 API and deprecated)

Synopsis

#include <pip.h>
int pip_spawn(char *filename, char **argv, char **envv, uint32_t coreno, int *pipidp, pip_spawnhook_t before, pip_spawnhook_t after, void *hookarg);

Description

This function spawns a PiP task.

In the process execution mode, the file descriptors having the `FD_CLOEXEC` flag is closed and will not be passed to the spawned PiP task. This simulated close-on-exec will not take place in the pthread execution mode.

Parameters

`in`	*filename*	The executable to run as a PiP task
`in`	*argv*	Argument(s) for the spawned PiP task
`in`	*envv*	Environment variables for the spawned PiP task
`in`	*coreno*	CPU core number for the PiP task to be bound to. By default, `coreno` is set to zero, for example, then the calling task will be bound to the first core available. This is in mind that the available core numbers are not contiguous. To specify an absolute core number, `coreno` must be bitwise-ORed with `PIP_CPUCORE_ABS`. If `PIP_CPUCORE_ASIS` is specified, then the core binding will not take place.
`in,out`	*pipidp*	Specify PiP ID of the spawned PiP task. If `PIP_PIPID_ANY` is specified, then the PiP ID of the spawned PiP task is up to the PiP library and the assigned PiP ID will be returned.
`in`	*before*	Just before the executing of the spawned PiP task, this function is called so that file descriptors inherited from the PiP root, for example, can deal with. This is only effective with the PiP process mode. This function is called with the argument *hookarg* described below.

in	*after*	This function is called when the PiP task terminates for the cleanup purpose. This function is called with the argument *hookarg* described below.
in	*hookarg*	The argument for the *before* and *after* function call.

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not yet initialized
EPERM	PiP task tries to spawn child task
EINVAL	`progp` is `NULL`
EINVAL	`opts` is invalid and/or unacceptable
EINVAL	the value off `pipidp` is invalid
EINVAL	the coreno is larger than or equal to `PIP_CPUCORE_CORENO_MAX`
EBUSY	specified PiP ID is alredy occupied
ENOMEM	not enough memory
ENXIO	`dlmopen` failss

Environment

- **PIP_STOP_ON_START** Specifying the PIP ID to stop on start PiP task program to debug from the beginning. If the before hook is specified, then the PiP task will be stopped just before calling the before hook.

Bugs

In theory, there is no reason to restrict for a PiP task to spawn another PiP task. However, the current glibc implementation does not allow to do so.

If the root process is multithreaded, only the main thread can call this function.

See Also

pip_task_spawn(3), pip_spawn_from_main(3), pip_spawn_from_func(3), pip_spawn_hook(3), pip_task_spawn(3), pip_blt_spawn(3)

3.2.1.7 pip_blt_spawn

spawn a PiP BLT/ULP (Bi-Level Task / User-Level Process)

Synopsis

#include <pip.h>
int pip_blt_spawn(pip_spawn_program_t *progp, uint32_t coreno, uint32_t opts, int *pipidp, pip_task_t **bltp, pip_task_queue_t *queue, pip_spawn_hook_t *hookp);

Description

This function spawns a BLT (PiP task) specified by `progp`. The created annd returned BLT is another form of a PiP task. It is an opaque object, essentially a double-linked list. Thus created BLT can be enqueued or dequeued to/from a `pip_task_queue_t`.

In the process execution mode, the file descriptors having the `FD_CLOEXEC` flag is closed and will not be passed to the spawned PiP task. This simulated close-on-exec will not take place in the pthread execution mode.

Parameters

out	*hook*	Pointer to the pip_spawn_hook_t structure in which the invocation hook information is set
in	*coreno*	CPU core number for the PiP task to be bound to. By default, coreno is set to zero, for example, then the calling task will be bound to the first core available. This is in mind that the available core numbers are not contiguous. To specify an absolute core number, coreno must be bitwise-ORed with PIP_CPUCORE_ABS. If PIP_CPUCORE_ASIS is specified, then the core binding will not take place.
in	*opts*	option flags. If PIP_TASK_INACTIVE is set, the created BLT is suspended and enqueued to the specified queue. Otherwise the BLT will schedules the BLTs in queue.
in,out	*pipidp*	Specify PiP ID of the spawned PiP task. If PIP_PIPID_ANY is specified, then the PiP ID of the spawned PiP task is up to the PiP library and the assigned PiP ID will be returned.
in,out	*bltp*	returns created BLT
in	*queue*	PiP task queue. See the above opts description.
in	*hookp*	Hook information to be invoked before and after the program invokation.

Note

In theory, there is no reason to restrict for a PiP task to spawn another PiP task. However, the current implementation fails to do so. If the root process is multithreaded, only the main thread can call this function. In the process mode, the file descriptors set the close-on-exec flag will be closed on the created child task.

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not yet initialized
EPERM	PiP task tries to spawn child task
EINVAL	progp is NULL
EINVAL	opts is invalid and/or unacceptable
EINVAL	the value off pipidp is invalid
EBUSY	specified PiP ID is alredy occupied
ENOMEM	not enough memory
ENXIO	dlmopen failss

Environment

- **PIP_STOP_ON_START** Specifying the PIP ID to stop on start PiP task program to debug from the beginning. If the before hook is specified, then the PiP task will be stopped just before calling the before hook.

Bugs

In theory, there is no reason to restrict for a PiP task to spawn another PiP task. However, the current glibc implementation does not allow to do so.

If the root process is multithreaded, only the main thread can call this function.

See Also

pip_task_spawn(3), pip_spawn_from_main(3), pip_spawn_from_func(3), pip_spawn_hook(3), pip_task_spawn(3), pip_spawn(3)

3.3 Terminating PiP Task

Functions

- void **pip_exit** (int status)
- int **pip_kill_all_tasks** (void)
- void **pip_abort** (void)

3.3.1 Detailed Description

3.3.1.1 Terminating PiP task

Description

Function to ternminate PiP task normally or abnormally (abort).

3.3.1.2 pip_exit

terminate the calling PiP task

Synopsis

#include <pip.h>
void pip_exit(int status);

Description

When the main function or the start function of a PiP task returns with an integer value, then it has the same effect of calling `pip_exit` with the returned value.

Parameters

`in`	*status*	This status is returned to PiP root.

Note

This function can be used regardless to the PiP execution mode. `exit(3)` is called in the process mode and `pthread_exit(3)` is called in the pthread mode.

See Also

pip_wait(3), pip_trywait(3), pip_wait_any(3), pip_trywait_any(3)

3.3.1.3 pip_kill_all_tasks

kill all PiP tasks

Synopsis

#include <pip.h>
int pip_kill_all_tasks(void);

Note

This function must be called from PiP root.

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	The PiP library is not initialized yet
EPERM	Not called from root

3.3.1.4 pip_abort

Kill all PiP tasks and then kill PiP root

Synopsis

```
#include <pip.h>
void pip_abort( void );
```

3.4 Waiting for PiP task termination

Functions

- int **pip_wait** (int pipid, int *status)
- int **pip_trywait** (int pipid, int *status)
- int **pip_wait_any** (int *pipid, int *status)
- int **pip_trywait_any** (int *pipid, int *retval)

3.4.1 Detailed Description

3.4.1.1 Waiting for PiP task termination

Description

Functions to wait for PiP task termination. All functions listed here must only be called from PiP root.

3.4.1.2 pip_wait

wait for the termination of a PiP task

Synopsis

```
#include <pip.h>
int pip_wait( int pipid, int *status );
```

Description

This function can be used regardless to the PiP execution mode. This function blocks until the specified PiP task terminates. The macros such as `WIFEXITED` and so on defined in Glibc can be applied to the returned `status` value.

Parameters

`in`	*pipid*	PiP ID to wait for.
`out`	*status*	Status value of the terminated PiP task

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not initialized yet
EPERM	This function is called other than PiP root
EDEADLK	The specified `pipid` is the one of PiP root
ECHILD	The target PiP task does not exist or it was already terminated and waited for

See Also

pip_exit(3), pip_trywait(3), pip_wait_any(3), pip_trywait_any(3)

3.4.1.3 pip_trywait

wait for the termination of a PiP task in a non-blocking way

Synopsis

```
#include <pip.h>
int pip_trywait( int pipid, int *status );
```

Description

This function can be used regardless to the PiP execution mode. This function behaves like the `wait` function of glibc and the macros such as `WIFEXITED` and so on can be applied to the returned `status` value.

Synopsis

```
#include <pip.h>
int pip_trywait( int pipid, int *status );
```

Parameters

`in`	*pipid*	PiP ID to wait for.
`out`	*status*	Status value of the terminated PiP task

Note

This function can be used regardless to the PiP execution mode.

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	The PiP library is not initialized yet
EPERM	This function is called other than PiP root
EDEADLK	The specified `pipid` is the one of PiP root
ECHILD	The target PiP task does not exist or it was already terminated and waited for

See Also

pip_exit(3), pip_wait(3), pip_wait_any(3), pip_trywait_any(3)

3.4.1.4 pip_wait_any

Wait for the termination of any PiP task

Synopsis

#include <pip.h>
int pip_wait_any(int *pipid, int *status);

Description

This function can be used regardless to the PiP execution mode. This function blocks until any of PiP tasks terminates. The macros such as `WIFEXITED` and so on defined in Glibc can be applied to the returned `status` value.

Parameters

`out`	*pipid*	PiP ID of terminated PiP task.
`out`	*status*	Exit value of the terminated PiP task

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	The PiP library is not initialized yet
EPERM	This function is called other than PiP root
ECHILD	The target PiP task does not exist or it was already terminated and waited for

See Also

pip_exit(3), pip_wait(3), pip_trywait(3), pip_trywait_any(3)

3.4.1.5 pip_trywait_any

non-blocking version of `pip_wait_any`

Synopsis

#include <pip.h>
int pip_trywait_any(int *pipid, int *status);

Description

This function can be used regardless to the PiP execution mode. This function blocks until any of PiP tasks terminates. The macros such as `WIFEXITED` and so on defined in Glibc can be applied to the returned `status` value.

Parameters

`out`	*pipid*	PiP ID of terminated PiP task.
`out`	*status*	Exit value of the terminated PiP task

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	The PiP library is not initialized yet
EPERM	This function is called other than PiP root
ECHILD	There is no PiP task to wait for

See Also

pip_exit(3), pip_wait(3), pip_trywait(3), pip_wait_any(3)

3.5 Export/Import Functions

Functions

- int **pip_named_export** (void ∗exp, const char ∗format,...) __attribute__((format(printf
- int **pip_named_import** (int pipid, void ∗∗expp, const char ∗format,...) __attribute__((format(printf
- int **pip_named_tryimport** (int pipid, void ∗∗expp, const char ∗format,...) __attribute__((format(printf
- int **pip_export** (void ∗exp)
- int **pip_import** (int pipid, void ∗∗expp)

3.5.1 Detailed Description

3.5.1.1 PiP Export and Import

Description

Export and import functions to exchange addresses among tasks

3.5.1.2 pip_named_export

export an address of the calling PiP root or a PiP task to the others.

Synopsis

#include <pip.h>
int pip_named_export(void ∗exp, const char ∗format, ...)

Description

Pass an address of a memory region to the other PiP task. Unlike the simmple `pip_export` and `pip_import` functions which can only export one address per task, `pip_named_export` and `pip_named_import` can associate a name with an address so that PiP root or PiP task can exchange arbitrary number of addressess.

Parameters

`in`	*exp*	an address to be passed to the other PiP task
`in`	*format*	a `printf` format to give the exported address a name. If this is `NULL`, then the name is assumed to be "".

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	`pip_init` is not yet called.
EBUSY	The name is already registered.
ENOMEM	Not enough memory

Note

The addresses exported by `pip_named_export` cannot be imported by calling `pip_import`, and vice versa.

See Also

pip_named_import(3)

3.5.1.3 pip_named_import

import the named exported address

Synopsis

#include <pip.h>
int pip_named_import(int pipid, void **expp, const char *format, ...)

Description

Import an address exported by the specified PiP task and having the specified name. If it is not exported yet, the calling task will be blocked. The

Parameters

`in`	*pipid*	The PiP ID to import the exposed address
`out`	*expp*	The starting address of the exposed region of the PiP task specified by the *pipid*.
`in`	*format*	a `printf` format to give the exported address a name

Note

There is possibility of deadlock when two or more tasks are mutually waiting for exported addresses.

The addresses exported by `pip_export` cannot be imported by calling `pip_named_import`, and vice versa.

Returns

zero is returned if this function succeeds. On error, an error number is returned.

Return values

EPERM	`pip_init` is not yet called.
EINVAL	The specified `pipid` is invalid
ENOMEM	Not enough memory

ECANCELED	The target task is terminated
EDEADLK	`pipid` is the calling task and tries to block itself

See Also

pip_named_export(3), pip_named_tryimport(3), pip_export(3), pip_import(3)

3.5.1.4 pip_named_tryimport

import the named exported address (non-blocking)

Synopsis

#include <pip.h>
int pip_named_tryimport(int pipid, void **expp, const char *format, ...)

Description

Import an address exported by the specified PiP task and having the specified name. If it is not exported yet, this returns `EAGAIN`.

Parameters

`in`	*pipid*	The PiP ID to import the exposed address
`out`	*expp*	The starting address of the exposed region of the PiP task specified by the *pipid*.
`in`	*format*	a `printf` format to give the exported address a name

Note

The addresses exported by `pip_export` cannot be imported by calling `pip_named_import`, and vice versa.

Returns

Zero is returned if this function succeeds. On error, an error number is returned.

Return values

EPERM	`pip_init` is not yet called.
EINVAL	The specified `pipid` is invalid
ENOMEM	Not enough memory
ECANCELED	The target task is terminated
EAGAIN	Target is not exported yet

See Also

pip_named_export(3), pip_named_import(3), pip_export(3), pip_import(3)

3.5.1.5 pip_export

export an address

Synopsis

#include <pip.h>
int pip_export(void *exp);

Description

Pass an address of a memory region to the other PiP task. This is a very naive implementation in PiP v1 and deprecated. Once a task export an address, there is no way to change the exported address or undo export.

Parameters

in	*exp*	An addresss

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not initialized yet

See Also

pip_import(3), pip_named_export(3), pip_named_import(3), pip_named_tryimport(3)

3.5.1.6 pip_import

import exported address of a PiP task

Synopsis

```
#include <pip.h>
int pip_export( void **expp );
```

Description

Get an address exported by the specified PiP task. This is a very naive implementation in PiP v1 and deprecated. If the address is not yet exported at the time of calling this function, then `NULL` is returned.

Parameters

in	*pipid*	The PiP ID to import the exportedaddress
out	*expp*	The exported address

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not initialized yet

See Also

pip_export(3), pip_named_export(3), pip_named_import(3), pip_named_tryimport(3)

3.6 PiP Miscellaneous Functions

Functions

- int **pip_get_pipid** (int *pipidp)
- int **pip_is_initialized** (void)
- int **pip_get_ntasks** (int *ntasksp)
- int **pip_get_mode** (int *modep)
- const char * **pip_get_mode_str** (void)
- int **pip_get_system_id** (int pipid, pip_id_t *idp)
- int **pip_isa_root** (void)
- int **pip_isa_task** (void)
- int **pip_is_threaded** (int *flagp)
- int **pip_is_shared_fd** (int *flagp)

3.6.1 Detailed Description

3.6.1.1 PiP miscellaneous functions

Description

Miscellaneous functions for PiP task (not BLT/ULP)

3.6.1.2 pip_get_pipid

get PiP ID of the calling task

Synopsis

```
#include <pip.h>
int pip_get_pipid( int *pipidp );
```

Parameters

out	*pipidp*	This parameter points to the variable which will be set to the PiP ID of the calling task

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not initialized yet

3.6.1.3 pip_is_initialized

Query is PiP library is already initialized

Synopsis

```
#include <pip.h>
int pip_is_initialized( void );
```

Returns

Return a non-zero value if PiP is already initialized. Otherwise this returns zero.

3.6.1.4 pip_get_ntasks

get the maximum number of the PiP tasks

Synopsis

```
#include <pip.h>
int pip_get_ntasks( int *ntasksp );
```

Parameters

out	*ntasksp*	Maximum number of PiP tasks is returned

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not yet initialized

3.6.1.5 pip_get_mode

get the PiP execution mode

Synopsis

```
#include <pip.h>
int pip_get_mode( int *modep );
```

Parameters

out	*modep*	Returned PiP execution mode

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not yet initialized

See Also

pip_get_mode_str(3)

3.6.1.6 pip_get_mode_str

get a character string of the current execution mode

Synopsis

```
#include <pip.h>
char *pip_get_mode_str( void );
```

Returns

Return the name string of the current execution mode. If PiP library is note initialized yet, then thiss return `NULL`.

3.6.1.7 pip_get_system_id

deliver a process or thread ID defined by the system

Synopsis

```
#include <pip.h>
int pip_get_system_id( int *pipid, uintptr_t *idp );
```

Description

The returned object depends on the PiP execution mode. In the process mode it returns TID (Thread ID, not PID) and in the thread mode it returns thread (`pthread_t`) associated with the PiP task This function can be used regardless to the PiP execution mode.

Parameters

`out`	*pipid*	PiP ID of a target PiP task
`out`	*idp*	a pointer to store the ID value

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	The PiP library is not initialized yet

3.6.1.8 pip_isa_root

check if calling PiP task is a PiP root or not

Synopsis

```
#include <pip.h>
int pip_isa_root( void );
```

Returns

Return a non-zero value if the caller is the PiP root. Otherwise this returns zero.

3.6.1.9 pip_isa_task

check if calling PiP task is a PiP task or not

Synopsis

```
#include <pip.h>
int pip_isa_task( void );
```

Returns

Return a non-zero value if the caller is the PiP task. Otherwise this returns zero.

3.6.1.10 pip_is_threaded

check if PiP execution mode is pthread or not

Synopsis

```
#include <pip.h>
int pip_is_threaded( int *flagp );
```

Parameters

out	*set*	to a non-zero value if PiP execution mode is Pthread

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	The PiP library is not initialized yet

3.6.1.11 pip_is_shared_fd

check if file descriptors are shared or not. This is equivalent with the `pip_is_threaded` function.

Synopsis

```
#include <pip.h>
int pip_is_shared_fd( int *flagp );
```

Parameters

out	*set*	to a non-zero value if FDs are shared

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	The PiP library is not initialized yet

3.7 PiP Signaling Functions

Functions

- int **pip_kill** (int pipid, int signal)
- int **pip_sigmask** (int how, const sigset_t *sigmask, sigset_t *oldmask)
- int **pip_signal_wait** (int signal)

3.7.1 Detailed Description

3.7.1.1 PiP signaling functions

Description

Signal manupilating functions. All functions listed here are agnostic to the PiP execution mode.

3.7.1.2 pip_kill

deliver a signal to PiP task

Synopsis

```
#include <pip.h>
int pip_kill( int pipid, int signal );
```

Parameters

out	*pipid*	PiP ID of a target PiP task to deliver the signal
out	*signal*	signal number to be delivered

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not yet initialized
EINVAL	An invalid signal number or invalid PiP ID is specified

See Also

tkill(2)

3.7.1.3 pip_sigmask

set signal mask of the current PiP task

Synopsis

#include <pip.h>
int pip_sigmask(int how, const sigset_t *sigmask, sigset_t *oldmask);

Description

This function is agnostic to the PiP execution mode.

Parameters

in	*how*	see **sigprogmask** or **pthread_sigmask**
in	*sigmask*	signal mask
out	*oldmask*	old signal mask

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not yet initialized
EINVAL	An invalid signal number or invalid PiP ID is specified

See Also

sigprocmask, pthread_sigmask

3.7.1.4 pip_signal_wait

wait for a signal

Synopsis

#include <pip.h>
int pip_signal_wait(int signal);

Description

This function is agnostic to the PiP execution mode.

Parameters

in	*signal*	signal to wait

Returns

Return 0 on success. Return an error code on error.

Note

This function does NOT return the `EINTR` error. This case is treated as normal return;

See Also

sigwait, sigsuspend

Chapter 4

BLT/ULP Functions

4.1 Yielding Functionns

Functions

- int **pip_yield** (int flag)
- int **pip_yield_to** (pip_task_t *task)

4.1.1 Detailed Description

4.1.1.1 Yielding functions

Description

Yielding execution of the calling BLT/ULP

4.1.1.2 pip_yield

Yield

Synopsis

#include <pip.h>
int pip_yield(int flag);

Parameters

in	*flag*	to specify the behavior of yielding. See below.

Returns

No context-switch takes place during the call, then this returns zero. If the context-switch to the other BLT happens, then this returns EINTR.

Fag

Parameters

PIP_YIELD_USER	If the calling task is scheduling PiP task(s) then the calling task switch to the next eligible-to-run BLT.
PIP_YIELD_SYSTEM	Regardless if the calling task is active or inactive, it calls `sched_yield`.
PIP_YIELD_DEFAULT	

See Also

pip_yield_to(3)

4.1.1.3 pip_yield_to

Yield to the specified PiP task

Synopsis

```
#include <pip.h>
int pip_yield( pip_task_t *task );
```

Description

Context-switch to the specified PiP task. If `task` is `NULL`, then this works the same as what `pip_yield(3)` does with `PIP_YIELD_DEFAULT`.

Parameters

`in`	*task*	Target PiP task to switch.

Returns

Return `Zero` or EINTR on success. Return an error code on error.

Return values

EPERM	PiP library is not yet initialized or already
EPERM	The specified task belongs to the other scheduling domain.

See Also

pip_yield(3)

4.2 Task Queue Operations

Functions

- int **pip_task_queue_init** (pip_task_queue_t *queue, pip_task_queue_methods_t *methods)
- int **pip_task_queue_trylock** (pip_task_queue_t *queue)
- void **pip_task_queue_lock** (pip_task_queue_t *queue)
- void **pip_task_queue_unlock** (pip_task_queue_t *queue)
- int **pip_task_queue_isempty** (pip_task_queue_t *queue)
- int **pip_task_queue_count** (pip_task_queue_t *queue, int *np)
- void **pip_task_queue_enqueue** (pip_task_queue_t *queue, pip_task_t *task)
- pip_task_t * **pip_task_queue_dequeue** (pip_task_queue_t *queue)
- void **pip_task_queue_describe** (pip_task_queue_t *queue, FILE *fp)
- int **pip_task_queue_fin** (pip_task_queue_t *queue)

4.2.1 Detailed Description

4.2.1.1 Task queue operations

Description

Manipulating ULP/BLT task queue functions

4.2.1.2 pip_task_queue_init

Initialize task queue

Synopsis

#include <pip.h>
int pip_task_queue_init(pip_task_queue_t *queue, pip_task_queue_methods_t *methods);

Parameters

in	*queue*	A task queue
in	*methods*	Must be set to NULL. Researved for future use.

Returns

Always return 0.

4.2.1.3 pip_task_queue_trylock

Try locking task queue

Synopsis

#include <pip.h>
int pip_task_queue_trylock(pip_task_queue_t *queue);

Parameters

in	*queue*	A task queue

Returns

Returns a non-zero value if lock succeeds.

4.2.1.4 pip_task_queue_lock

Lock task queue

Synopsis

#include <pip.h>
int pip_task_queue_lock(pip_task_queue_t *queue);

Parameters

in	*queue*	A task queue

Returns

This function returns no error

4.2.1.5 pip_task_queue_unlock

Unlock task queue

Synopsis

```
#include <pip.h>
int pip_task_queue_unlock( pip_task_queue_t *queue );
```

Parameters

in	*queue*	A task queue

Returns

This function returns no error

4.2.1.6 pip_task_queue_isempty

Query function if the current task has some tasks to be scheduled with.

Synopsis

```
#include <pip.h>
int pip_task_queue_isempty( pip_task_queue_t *queue );
```

Parameters

in	*queue*	A task queue

Returns

Returns a non-zero value if the queue is empty

4.2.1.7 pip_task_queue_count

Count the length of task queue

Synopsis

```
#include <pip.h>
int pip_task_queue_count( pip_task_queue_t *queue, int *np );
```

Parameters

in	*queue*	A task queue
out	*np*	the queue length returned

Returns

Return 0 on success. Return an error code on error.

Return values

EINVAL	`queue` is `NULL`
EINVAL	`np` is `NULL`

4.2.1.8 pip_task_queue_enqueue

Enqueue a BLT

Synopsis

#include <pip.h>
void pip_task_queue_enqueue(pip_task_queue_t *queue, pip_task_t *task);

Parameters

in	*queue*	A task queue
in	*task*	A task to be enqueued

Note

It is the user responsibility to lock (and unlock) the queue.

4.2.1.9 pip_task_queue_dequeue

Dequeue a task from a task queue

Synopsis

#include <pip.h>
pip_task_t* pip_task_queue_dequeue(pip_task_queue_t *queue);

Parameters

in	*queue*	A task queue

Returns

Dequeued task iss returned. If the queue is empty then `NULL` is returned.

Note

It is the user responsibility to lock (and unlock) the queue.

4.2.1.10 pip_task_queue_describe

Describe queue

Synopsis

#include <pip.h>
void pip_task_queue_describe(pip_task_queue_t *queue, FILE *fp);

Parameters

in	*queue*	A task queue
in	*fp*	a File pointer

4.2.1.11 pip_task_queue_fin

Finalize a task queue

Synopsis

```
#include <pip.h>
int pip_task_queue_fin( pip_task_queue_t *queue );
```

Parameters

in	*queue*	A task queue

Returns

Zero is returned always

4.3 Suspending and Resuming BLT/ULP

Functions

- int **pip_suspend_and_enqueue** (pip_task_queue_t *queue, pip_enqueue_callback_t callback, void *cbarg)
- int **pip_suspend_and_enqueue_nolock** (pip_task_queue_t *queue, pip_enqueue_callback_t callback, void *cbarg)
- int **pip_dequeue_and_resume** (pip_task_queue_t *queue, pip_task_t *sched)
- int **pip_dequeue_and_resume_nolock** (pip_task_queue_t *queue, pip_task_t *sched)
- int **pip_dequeue_and_resume_N** (pip_task_queue_t *queue, pip_task_t *sched, int *np)
- int **pip_dequeue_and_resume_N_nolock** (pip_task_queue_t *queue, pip_task_t *sched, int *np)

4.3.1 Detailed Description

4.3.1.1 Suspending and resuming BLT/ULP

Description

Suspending and resuming BLT/ULP

4.3.1.2 pip_suspend_and_enqueue

suspend the curren task and enqueue it with lock

Synopsis

```
#include <pip.h>
int pip_suspend_and_enqueue( pip_task_queue_t *queue, pip_enqueue_callback_t callback, void *cbarg );
```

Description

The **queue** is locked just before the calling task is enqueued and unlocked after the calling task is enqueued. After then the **callback** function is called.

As the result of this suspension, a context-switch takes place if there is at least one elgible-to-run task in the scheduling queue (this is hidden from users). If there is no other task to schedule then the kernel thread of the current task will be blocked.

Parameters

in	*queue*	A task queue
in	*callback*	A callback function which is called immediately after the task is enqueued
in	*cbarg*	An argument given to the callback function

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not initialized yet
EINVAL	`queue` is `NULL`

See Also

pip_enqueu_and_suspend_nolock(3), pip_dequeue_and_resume(3)

4.3.1.3 pip_suspend_and_enqueue_nolock

suspend the curren task and enqueue it without locking the queue

Synopsis

#include <pip.h>
int pip_suspend_and_enqueue_nolock(pip_task_queue_t *queue, pip_enqueue_callback_t callback, void *cbarg);

Description

Unlike `pip_suspend_and_enqueue`, this function never locks the queue. It is the user's responsibility to lock the queue beofre calling this function and unlock the queue after calling this function. The **callback** function can be used for unlocking.

As the result of this suspension, a context-switch takes place if there is at least one elgible-to-run task in the scheduling queue (this is hidden from users). If there is no other task to schedule then the kernel thread of the current task will be blocked.

Parameters

in	*queue*	A task queue
in	*callback*	A callback function which is called when enqueued
in	*cbarg*	An argument given to the callback function

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not initialized yet
EINVAL	`queue` is `NULL`

4.3.1.4 pip_dequeue_and_resume

dequeue a task and make it runnable

Description

The **queue** is locked and then unlocked when to dequeued a task.

Synopsis

#include <pip.h>
int pip_dequeue_and_resume(pip_task_queue_t *queue, pip_task_t *sched);

Parameters

`in`	*queue*	A task queue
`in`	*sched*	A task to specify a scheduling domain

Returns

If succeedss, 0 is returned. Otherwise an error code is returned.

Return values

EPERM	PiP library is not initialized yet
EINVAL	`queue` is `NULL`
ENOENT	`queue` is empty.

4.3.1.5 pip_dequeue_and_resume_nolock

dequeue a task and make it runnable

Synopsis

#include <pip.h>
int pip_dequeue_and_resume(pip_task_queue_t *queue, pip_task_t *sched);

Description

Task in the queue is dequeued and scheduled by the specified `sched`. If `sched` is NULL, then the task is enqueued into the scheduling queue of calling task.

It is the user's responsibility to lock the queue beofre calling this function and unlock the queue after calling this function.

Parameters

in	*queue*	A task queue
in	*sched*	A task to specify a scheduling domain

Returns

This function returns no error

Return values

EPERM	PiP library is not initialized yet
EINVAL	queue is NULL
ENOENT	queue is empty.

4.3.1.6 pip_dequeue_and_resume_N

dequeue multiple tasks and resume the execution of them

Synopsis

#include <pip.h>
int pip_dequeue_and_resume_N(pip_task_queue_t *queue, pip_task_t *sched, int *np);

Description

The specified number of tasks are dequeued and scheduled by the specified sched. If sched is NULL, then the task is enqueued into the scheduling queue of calling task.

The **queue** is locked and unlocked when dequeued.

Parameters

in	*queue*	A task queue
in	*sched*	A task to specify a scheduling domain
in,out	*np*	A pointer to an interger which spcifies the number of tasks dequeued and actual number of tasks dequeued is returned. When PIP_TASK_ALL is specified, then all tasks in the queue will be resumed.

Returns

This function returns no error

Return values

EPERM	PiP library is not initialized yet
EINVAL	queue is NULL
EINVAL	the specified number of tasks is invalid
ENOENT	queue is empty.

It is the user's responsibility to lock the queue beofre calling this function and unlock the queue after calling this function.

4.3.1.7 pip_dequeue_and_resume_N_nolock

dequeue tasks and resume the execution of them

Synopsis

#include <pip.h>
int pip_dequeue_and_resume_N_nolock(pip_task_queue_t *queue, pip_task_t *sched, int *np);

Description

The specified number of tasks are dequeued and scheduled by the specified `sched`. If `sched` is NULL, then the task is enqueued into the scheduling queue of calling task.

It is the user's responsibility to lock the queue beofre calling this function and unlock the queue after calling this function.

Parameters

`in`	*queue*	A task queue
`in`	*sched*	A task to specify a scheduling domain
`in,out`	*np*	A pointer to an interger which spcifies the number of tasks dequeued and actual number of tasks dequeued is returned. When `PIP_TASK_ALL` is specified, then all tasks in the queue will be resumed.

Returns

This function returns no error

Return values

EPERM	PiP library is not initialized yet
EINVAL	`queue` is `NULL`
EINVAL	the specified number of tasks is invalid
ENOENT	`queue` is empty.

4.4 BLT/ULP Barrier Functions

Functions

- int **pip_barrier_init** (pip_barrier_t ∗barrp, int n)
- int **pip_barrier_wait** (pip_barrier_t ∗barrp)
- int **pip_barrier_fin** (pip_barrier_t ∗barrp)

4.4.1 Detailed Description

4.4.1.1 BLT/ULP barrier synchronization functions

Description

BLT/ULP barrier synchronization functions

Description

BLT/ULP mutex functions

Description

BLT/ULP coupling/decoupling functions

4.4.1.2 pip_barrier_init

initialize barrier synchronization structure

Synopsis

#include <pip.h>
int pip_barrier_init(pip_barrier_t *barrp, int n);

Parameters

in	*barrp*	pointer to a PiP barrier structure
in	*n*	number of participants of this barrier synchronization

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not yet initialized or already finalized
EINAVL	n is invalid

Note

This barrier works on PiP tasks only.

See Also

pip_barrier_init(3), pip_barrier_fin(3),

4.4.1.3 pip_barrier_wait

wait on barrier synchronization in a busy-wait way int pip_barrier_wait(pip_barrier_t *barrp);

Synopsis

#include <pip.h>
int pip_barrier_wait(pip_barrier_t *barrp);

Parameters

in	*barrp*	pointer to a PiP barrier structure

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not yet initialized or already finalized

See Also

pip_barrier_init(3), pip_barrier_fin(3),

4.4.1.4 pip_barrier_fin

finalize barrier synchronization structure

Synopsis

#include <pip.h>
int pip_barrier_fin(pip_barrier_t *barrp);

Parameters

in	*barrp*	pointer to a PiP barrier structure

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not yet initialized or already finalized
EBUSY	there are some tasks wating for barrier synchronization

See Also

pip_barrier_init(3), pip_barrier_wait(3),

4.5 BLT/ULP Mutex Functions

Functions

- int **pip_mutex_init** (pip_mutex_t ∗mutex)
- int **pip_mutex_lock** (pip_mutex_t ∗mutex)
- int **pip_mutex_unlock** (pip_mutex_t ∗mutex)
- int **pip_mutex_fin** (pip_mutex_t ∗mutex)

4.5.1 Detailed Description

4.5.1.1 pip_mutex_init

Initialize PiP mutex

Synopsis

#include <pip.h>
int pip_mutex_init(pip_mutex_t ∗mutex);

Parameters

in,out	*mutex*	pointer to the PiP task mutex

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not yet initialized or already finalized

See Also

pip_mutex_lock(3), pip_mutex_unlock(3), pip_mutex_fin(3)

4.5.1.2 pip_mutex_lock

Lock PiP mutex

Synopsis

```
#include <pip.h>
int pip_mutex_lock( pip_mutex_t *mutex );
```

Parameters

in	*mutex*	pointer to the PiP task mutex

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not yet initialized or already finalized

See Also

pip_mutex_init(3), pip_mutex_unlock(3), pip_mutex_fin(3)

4.5.1.3 pip_mutex_unlock

Unlock PiP mutex

Synopsis

```
#include <pip.h>
int pip_mutex_unlock( pip_mutex_t *mutex );
```

Parameters

in	*mutex*	pointer to the PiP task mutex

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not yet initialized or already finalized

See Also

pip_mutex_init(3), pip_mutex_lock(3), pip_mutex_fin(3)

4.5.1.4 pip_mutex_fin

Finalize PiP mutex

Synopsis

```
#include <pip.h>
int pip_mutex_fin( pip_mutex_t *mutex );
```

Parameters

in,out	*mutex*	pointer to the PiP task mutex

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not yet initialized or already finalized
EBUSY	There is one or more waiting PiP task

See Also

pip_mutex_lock(3), pip_mutex_unlock(3)

4.6 BLT/ULP Coupling/Decoupling Functions

Functions

- int **pip_couple** (void)
- int **pip_decouple** (pip_task_t *task)

4.6.1 Detailed Description

4.6.1.1 pip_couple

Couple the curren task with the original kernel thread

Synopsis

```
#include <pip.h>
int pip_couple( void );
```

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not yet initialized or already finalized
EBUSY	the curren task is already coupled with a kernel thread

4.6.1.2 pip_decouple

Decouple the curren task from the kernel thread

Synopsis

```
#include <pip.h>
int pip_decouple( pip_task_t *sched )
```

Parameters

in	*task*	specify the scheduling task to schedule the decoupled task (calling this function). If NULL, then the previously coupled pip_task takes place.

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not yet initialized or already finalized
EBUSY	the curren task is already decoupled from a kernel thread

4.7 BLT/ULP Miscellaneous Function

Functions

- pip_task_t * **pip_task_self** (void)
- int **pip_get_task_pipid** (pip_task_t *task, int *pipidp)
- int **pip_get_task_by_pipid** (int pipid, pip_task_t **taskp)
- int **pip_set_aux** (pip_task_t *task, void *aux)
- int **pip_get_aux** (pip_task_t *task, void **auxp)
- int **pip_get_sched_domain** (pip_task_t **domainp)

4.7.1 Detailed Description

4.7.1.1 BLT/ULP miscellaneous function

Description

BLT/ULP miscellaneous function

4.7.1.2 pip_task_self

Return the current task

Synopsis

```
#include <pip.h>
pip_task_t *pip_task_self( void );
```

Returns

Return the current task.

4.7.1.3 pip_get_task_pipid

Return PIPID of a PiP task

Synopsis

```
#include <pip.h>
int pip_get_task_pipid( pip_task_t *task, int *pipidp );
```

Parameters

in	*task*	a PiP task
out	*pipidp*	PiP ID of the specified task

Returns

Return 0 on success. Return an error code on error.

Return values

EINAVL	`task` is `NULL`
EPERM	PiP library is not yet initialized or already finalized

4.7.1.4 pip_get_task_by_pipid

get PiP task from PiP ID

Synopsis

```
#include <pip.h>
int pip_get_task_by_pipid( int pipid, pip_task_t **taskp );
```

Parameters

in	*pipid*	PiP ID
out	*taskp*	returning PiP task of the specified PiP ID

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not yet initialized or already finalized
ENOENT	No such PiP task
ERANGE	The specified `pipid` is out of ramge

4.7.1.5 pip_set_aux

Associate user data with a PiP task

Synopsis

```
#include <pip.h>
int pip_set_aux( pip_task_t *task, void *aux );
```

Parameters

in	*task*	PiP task. If `NULL`, then the data is associated with the current PiP task
in	*aux*	Pointer to the user dat to assocate with

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not yet initialized or already finalized

See Also

pip_get_aux(3)

4.7.1.6 pip_get_aux

Retrive the user data associated with a PiP task

Synopsis

```
#include <pip.h>
int pip_get_aux( pip_task_t *task, void **auxp );
```

Parameters

`in`	*task*	PiP task. If `NULL`, then the data is associated with the current PiP task
`out`	*auxp*	Returned user data

Returns

Return 0 on success. Return an error code on error.

Return values

EINAVL	`domainp` is `NULL` or `auxp` is `NULL`
EPERM	PiP library is not yet initialized or already finalized

See Also

pip_set_aux(3)

4.7.1.7 pip_get_sched_domain

Return the task representing the scheduling domain

Synopsis

```
#include <pip.h>
int pip_get_sched_domain( pip_task_t **domainp );
```

Parameters

`out`	*domainp*	Returned scheduling domain of the current task

Returns

Return 0 on success. Return an error code on error.

Return values

EPERM	PiP library is not yet initialized or already finalized

Index