



Processs-in-Process (PiP)

2.0.0

Refernce Manual

September 29, 2020

Generated by Doxygen 1.8.5

Contents

1	Proces-in-Process (PiP) Overview	1
2	PiP Commands	9
2.1	pipcc	9
2.2	pip-check	9
2.3	pip-exec	10
2.4	pip-man	10
2.5	pip-mode	10
2.6	pips	12
2.7	printpipmode	12
3	PiP Functions	13
3.1	PiP Initialization/Finalization	13
3.1.1	Detailed Description	13
3.1.1.1	PiP Initialization/Finalization	13
3.1.2	Function Documentation	13
3.1.2.1	pip_init	13
3.1.2.2	pip_fin	15
3.2	Spawning PiP task	16
3.2.1	Detailed Description	16
3.2.1.1	PiP Spawng PiP (ULP/BLT) task	16
3.2.2	Function Documentation	16
3.2.2.1	pip_spawn_from_main	16
3.2.2.2	pip_spawn_from_func	17
3.2.2.3	pip_spawn_hook	17
3.2.2.4	pip_task_spawn	18
3.2.2.5	pip_spawn	20
3.3	Export/Import Functions	21
3.3.1	Detailed Description	21
3.3.1.1	PiP Export and Import	21
3.3.2	Function Documentation	21
3.3.2.1	pip_named_export	21
3.3.2.2	pip_named_import	22
3.3.2.3	pip_named_tryimport	23
3.3.2.4	pip_export	24
3.3.2.5	pip_import	25
3.4	Waiting for PiP task termination	25
3.4.1	Detailed Description	25
3.4.1.1	Waiting for PiP task termination	25
3.4.2	Function Documentation	26
3.4.2.1	pip_wait	26
3.4.2.2	pip_trywait	26
3.4.2.3	pip_wait_any	27
3.4.2.4	pip_trywait_any	28
3.5	PiP Query Functions	28
3.5.1	Detailed Description	29

3.5.1.1	PiP Query functions	29
3.5.2	Function Documentation	29
3.5.2.1	pip_get_pipid	29
3.5.2.2	pip_is_initialized	30
3.5.2.3	pip_get_ntasks	30
3.5.2.4	pip_get_mode	31
3.5.2.5	pip_get_mode_str	31
3.5.2.6	pip_get_system_id	31
3.5.2.7	pip_isa_root	32
3.5.2.8	pip_isa_task	32
3.5.2.9	pip_is_threaded	33
3.5.2.10	pip_is_shared_fd	33
3.6	PiP task termination	33
3.6.1	Detailed Description	34
3.6.1.1	Terminating PiP task	34
3.6.2	Function Documentation	34
3.6.2.1	pip_exit	34
3.6.2.2	pip_kill_all_tasks	35
3.6.2.3	pip_abort	35
3.6.2.4	pip_kill	35
3.7	PiP Siganling Functions	36
3.7.1	Detailed Description	36
3.7.1.1	PiP signaling functions	36
3.7.2	Function Documentation	36
3.7.2.1	pip_sigmask	36
3.7.2.2	pip_signal_wait	37
3.8	PiP Synchronization Functions	37
3.8.1	Detailed Description	38
3.8.1.1	PiP synchronization functions	38
3.8.2	Function Documentation	38
3.8.2.1	pip_yield	38
3.8.2.2	pip_barrier_init	38
3.8.2.3	pip_barrier_wait	39
3.8.2.4	pip_barrier_fin	39

Chapter 1

Process-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 owned by a process can be accessed by the other processes. Those processes share the same address space, just like pthreads, but each process has its own variables like the process execution model. Hereinafter, 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 become a ULT by decoupling the associated kernel thread from the KLT. The decoupled kernel thread becomes 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 means 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 own file descriptor space, for example. When a ULP is scheduled by a KLT having PID 1000, then the getpid() is called by the ULP returns 1000. Further, when the ULT is migrated to be scheduled by the other KLT, then the returned PID is different. So, when implementing a ULP system, this syscall 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 noted 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;

- Process and
- (P)Thread.

In the pthread mode, although each PiP task has its own static variables unlike thread, PiP task behaves more like PThread, 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, a 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 is set to `"(p)thread"` then the PiP library runs in the pthread mode, and if it is set to `"process"` then it runs in the process mode. There are also three implementations in the process mode; `"process:preload"`, `"process:pipeclone"` 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:pipeclone"` mode is only effective with the PiP-patched glibc library (see below).

Several functions are made available by the PiP library to absorb the functional differences due to the execution modes.

License

This package is licensed under the 2-clause simplified BSD License - see the `[LICENSE]`(`LICENSE`) file for details.

Installation

There are several ways to install PiP; Docker, Spack, RPM, and building from the source code.

Docker image

Download and run the PiP Docker image.

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

Spack

Download spack and do the following;

```
$ git clone https://github.com/spack/spack.git
$ cd spack/bin
$ spack install process-in-process
```

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** will be `"/usr."`

Source Code

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

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

By using PiP-glibc, users can create up to 300 PiP tasks which can be debugged by using PiP-gdb. In other words, without installing PiP-glibc, users can create up to around 10 PiP tasks (the number depends on the program) and cannot debug by using PiP-gdb. Above Docker image and Spack include PiP-glibc and PiP-gdb.

- **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_INSTALL_DIR
```

2. Build PiP library

The same source code can be used 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 doc
```

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

```
$ cd PIP_SRC_DIR $ make check-installed
```

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
```

CentOS8 or RHEL8

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

Build PiP-gdb

```
$ cd GDB_SRC_DIR $ ./build.sh --prefix=GDB_INSTALL_DIR --with-pip=PIP_INSTALL_DIR
```

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 examples will show you the same document you are reading.

PDF

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

HTML

HTML documents will be installed at **PIP_INSTALL_DIR**/share/html.

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 that you have a non-PiP program(s) and want 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. Note that the 'myprog.c' may or may not call any PiP functions. Your program can also run as a normal program (not as a PiP task) without using the pip-exec(1) command.

You may write your own PiP programs which 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 see how your PiP program is running in realtime 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 and PiP root which created those tasks. Each PiP 'processes' is treated as a GDB inferior in PiP-gdb.

```
$ 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 exception 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 specified or default signals is delivered, then PiP-gdb will be attached automatically. 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 PiP-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.

Mailing List

pip@ml.riken.jp

Publications

Research papers

Atsushi Hori, Min Si, Balazs Gerofi, Masamichi Takagi, Jay Dayal, Pavan Balaji, and Yutaka 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>

Atsushi Hori, Balazs Gerofi, and Yuataka Ishikawa. "An Implementation of User-Level Processes using Address Space Sharing," 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW),

New Orleans, LA, USA, 2020, pp. 976-984, DOI: <https://doi.org/10.1109/IPDPSW50202.2020.00161>.

Kaiming Ouyang, Min Si, Atsushi Hori, Zizhong Chen and Pavan Balaji. "CAB-MPI: Exploring Interprocess Work Stealing toward Balanced MPI Communication," in SC'20 (to appear)

Commands

- pipcc
- pip-check
- pip-exec
- pip-man
- pip-mode
- pips
- printpipmode

Functions

- PiP-0-init-fin
- PiP-1-spawn
- PiP-2-export
- PiP-3-wait
- PiP-4-query
- PiP-5-exit
- PiP-6-signal
- PiP-7-sync
- pip_abort
- pip_barrier_fin
- pip_barrier_init
- pip_barrier_wait
- pip_exit
- pip_export
- pip_fin
- pip_get_mode
- pip_get_mode_str
- pip_get_ntasks
- pip_get_pipid
- pip_get_system_id
- 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_named_export`
- `pip_named_import`
- `pip_named_tryimport`
- `pip_sigmask`
- `pip_signal_wait`
- `pip_spawn`
- `pip_spawn_from_func`
- `pip_spawn_from_main`
- `pip_spawn_hook`
- `pip_task_spawn`
- `pip_trywait`
- `pip_trywait_any`
- `pip_wait`
- `pip_wait_any`
- `pip_yield`

Author

Atsushi Hori

Riken Center for Computational 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

<i>-piproot</i>	the compile (and link) as a PiP root
<i>-piptask</i>	the compile (and link) as a PiP task
<i>-nopip</i>	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
pip-mode

2.2 pip-check

PiP binary checking program.

Synopsis

```
pip-check [OPTIONS] PIP-PROG [...]
```

Parameters

<i>-r</i>	check if a.out can be PiP root
<i>-t</i>	check if a.out can be PiP task
<i>-b</i>	check if a.out can be PiP root and/or PiP task (default)
<i>-v</i>	show reson

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

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

See Also

pipcc(1)

2.4 pip-man

show PiP man page

Synopsis

```
SYNOPSIS pipman [MAN_OPTS] MAN_TOPIC
```

See Also

man(1)

2.5 pip-mode

Set PiP execution mode

Synopsis

```
pip-mode [OPTION] [PIP-COMMAND]
```

Description

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

Parameters

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

See Also

`pip-exec`
`printpipmode`

2.6 pips

List or kill running PiP tasks.

Synopsis

`pips [OPTIONS] [PIP-COMMAND ...]`

Parameters

<code>-s</code>	Send the specified signal (followed by the "-b" option) to the specified PiP tasks
<code>-k</code>	same as -s
<code>-l</code>	List (using the Linux <code>ps</code> command) running PiP tasks specified. This is the default action.
<code>-t</code>	Show running PiP tasks specified by using the <code>top</code> command. Due to the <code>top</code> command limitation, only 20 PiP tasks will be shown.

See Also

`pip-exec`

2.7 printpipmode

command to print current PiP mode

Synopsis

`printpipmode`

Description

This command prints the current PiP mode setting

See Also

`pip-mode`

Chapter 3

PiP Functions

3.1 PiP Initialization/Finalization

Functions

- int [pip_init](#) (int *pipidp, int *ntasks, void **root_expp, int opts)
- int [pip_fin](#) (void)

3.1.1 Detailed Description

3.1.1.1 PiP Initialization/Finalization

Description

PiP initialization/finalization functions

3.1.2 Function Documentation

3.1.2.1 int pip_init(int * *pipidp*, int * *ntasks*, void ** *root_expp*, int *opts*)

Name

`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.

Description

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.

Description

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	<i>pipidp</i>	When this is called by the PiP root process, then this returns <code>PIP_PIPID_ROOT</code> , otherwise it returns the PiP ID of the calling PiP task.
in, out	<i>ntasks</i>	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	<i>root_expp</i>	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	<i>opts</i>	Specifying the PiP execution mode and See below.

Execution mode option

Users may explicitly specify the PiP execution mode. This execution mode can be categorized in two; process mode and thread mode. 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 execution mode, file descriptors and signal handlers are shared among PiP root and PiP tasks while maintaining the privatized variables.

To spawn a PiP task in the process mode, the PiP library modifies the `clone()` flag so that the created PiP task can exhibit the almost same way with that of normal Linux process. There are three ways implemented; using `LD_PRELOAD`, modifying Glibc, and modifying `GIOT` entry of the `clone()` systemcall. One of the option flag values; **PIP_MODE_PTHREAD**, **PIP_MODE_PROCESS**, **PIP_MODE_PROCESS_PRELOAD**, **PIP_MODE_PROCESS_PIPCLONE**, or **PIP_MODE_PROCESS_GOT** can be specified as the option flag. Or, users may specify the execution mode by the **PIP_MODE** environment described below.

Returns

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

Return values

<i>EINVAL</i>	<i>ntasks</i> is negative
<i>EBUSY</i>	PiP root called this function twice or more without calling pip_fin .
<i>EPERM</i>	<i>opts</i> is invalid or unacceptable
<i>EOVERFLOW</i>	<i>ntasks</i> is too large
<i>ELIBSCN</i>	version miss-match between PiP root and PiP task

Environment

- **PIP_MODE** Specifying the PiP execution mode. 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 the `pip_preload.so` file, if the PiP execution mode is `PIP_MODE_PROCESS_PRELOAD` (the `opts` in `pip_init`) and/or the `PIP_MODE` environment is set to `process:preload`. See also the `pip_mode` command to set the environment variable appropriately and easily.
- **PIP_STACKSZ** Specifying the stack size (in bytes). The **KMP_STACKSIZE** and **OMP_STACKSIZE** are also effective. The 't', 'g', 'm', 'k' and 'b' postfix character can be used.
- **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.

- **PIP_GDB_PATH** If this environment is set to the path pointing to the PiP-gdb executable file, then PiP-gdb is automatically attached when an execution signal (SIGSEGV and SIGHUP by default) is delivered. The signals which triggers the PiP-gdb invocation 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 execution signals is delivered, then the memory map will be shown.
- **PIP_SHOW_PIPS** If the value is 'on' and one of the above execution signals is delivered, then the process status by using the `pips` command (see also [pips](#)) will be shown.

Bugs

It 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](#)
[pip_export](#)
[pip_fin](#)
[pip-mode](#)
[pips](#)

3.1.2.2 int pip_fin(void)

Name

`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

<i>EPERM</i>	<code>pip_init</code> is not yet called
<i>EBUSY</i>	one or more PiP tasks are not yet terminated

Notes

The behavior of calling `pip_init` after calling this `pip_fin` is not defined and recommended to do so.

See Also

`pip_init`

3.2 Spawning PiP task

Functions

- void `pip_spawn_from_main` (`pip_spawn_program_t` *progp, char *prog, char **argv, char **envv, void *exp)
Setting information to invoke a PiP task starting from the main function.
- void `pip_spawn_from_func` (`pip_spawn_program_t` *progp, char *prog, char *funcname, void *arg, char **envv, void *exp)
Setting information to invoke a PiP task starting from a function defined in a program.
- void `pip_spawn_hook` (`pip_spawn_hook_t` *hook, `pip_spawnhook_t` before, `pip_spawnhook_t` after, void *hookarg)
Setting invocation hook information.
- int `pip_task_spawn` (`pip_spawn_program_t` *progp, `uint32_t` coreno, `uint32_t` opts, int *pipidp, `pip_spawn_hook_t` *hookp)
Spawning a PiP task.
- int `pip_spawn` (char *filename, char **argv, char **envv, int coreno, int *pipidp, `pip_spawnhook_t` before, `pip_spawnhook_t` after, void *hookarg)
spawn a PiP task (PiP v1 API and deprecated)

3.2.1 Detailed Description

3.2.1.1 PiP Spawng PiP (ULP/BLT) task

Description

Spawning PiP task or ULP/BLT task

3.2.2 Function Documentation

3.2.2.1 void pip_spawn_from_main (pip_spawn_program_t * progp, char * prog, char ** argv, char ** envv, void * exp)

Name

`pip_spawn_from_main`

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	<i>progp</i>	Pointer to the <code>pip_spawn_program_t</code> structure in which the program invocation information will be set
in	<i>prog</i>	Path to the executable file.
in	<i>argv</i>	Argument vector.
in	<i>envv</i>	Environment variables. If this is <code>NULL</code> , then the <code>environ</code> variable is used for the spawning PiP task.
in	<i>exp</i>	Export value to the spawning PiP task

See Also

[pip_task_spawn](#)
[pip_spawn_from_func](#)

3.2.2.2 `void pip_spawn_from_func (pip_spawn_program_t * progp, char * prog, char * funcname, void * arg, char ** envv, void * exp)`

Name

`pip_spawn_from_func`

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	<i>progp</i>	Pointer to the <code>pip_spawn_program_t</code> structure in which the program invocation information will be set
in	<i>prog</i>	Path to the executable file.
in	<i>funcname</i>	Function name to be started
in	<i>arg</i>	Argument which will be passed to the start function
in	<i>envv</i>	Environment variables. If this is <code>NULL</code> , then the <code>environ</code> variable is used for the spawning PiP task.
in	<i>exp</i>	Export value to the spawning PiP task

See Also

[pip_task_spawn](#)
[pip_spawn_from_main](#)

3.2.2.3 `void pip_spawn_hook (pip_spawn_hook_t * hook, pip_spawnhook_t before, pip_spawnhook_t after, void * hookarg)`

Name

`pip_spawn_hook`

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	<i>hook</i>	Pointer to the <code>pip_spawn_hook_t</code> structure in which the invocation hook information will be set
in	<i>before</i>	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 <i>hookarg</i> described below.
in	<i>after</i>	This function is called when the PiP task terminates for the cleanup purpose. This function is called with the argument <i>hookarg</i> described below.
in	<i>hookarg</i>	The argument for the <i>before</i> and <i>after</i> 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](#)

3.2.2.4 `int pip_task_spawn (pip_spawn_program_t *progp, uint32_t coreno, uint32_t opts, int *pipidp, pip_spawn_hook_t *hookp)`

Name

`pip_task_spawn`

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	<i>progp</i>	Pointer to the <code>pip_spawn_hook_t</code> structure in which the invocation hook information is set
in	<i>coreno</i>	CPU core number for the PiP task to be bound to. By default, <code>coreno</code> 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, <code>coreno</code> must be bitwise-ORed with <code>PIP_CPUCORE_ABS</code> . If <code>PIP_CPUCORE_ASIS</code> is specified, then the core binding will not take place.
in	<i>opts</i>	option flags
in, out	<i>pipidp</i>	Specify PiP ID of the spawned PiP task. If <code>PIP_PIPID_ANY</code> 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	<i>hookp</i>	Hook information to be invoked before and after the program invocation.

Returns

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

Return values

<i>EPERM</i>	PiP library is not yet initialized
<i>EPERM</i>	PiP task tries to spawn child task
<i>EINVAL</i>	<code>progp</code> is NULL
<i>EINVAL</i>	<code>opts</code> is invalid and/or unacceptable
<i>EINVAL</i>	the value of <code>pipidp</code> is invalid
<i>EINVAL</i>	the <code>coreno</code> is larger than or equal to <code>PIP_CPUCORE_CORENO_MAX</code>
<i>EBUSY</i>	specified PiP ID is already occupied
<i>ENOMEM</i>	not enough memory
<i>ENXIO</i>	<code>dlopen</code> fails

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 execution 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](#)
[pip_spawn_from_main](#)
[pip_spawn_from_func](#)
[pip_spawn_hook](#)
[pip_spawn](#)

```
3.2.2.5 int pip_spawn ( char * filename, char ** argv, char ** envv, int coreno, int * pipidp, pip_spawnhook_t before,
pip_spawnhook_t after, void * hookarg )
```

Name

pip_spawn

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	<i>filename</i>	The executable to run as a PiP task
in	<i>argv</i>	Argument(s) for the spawned PiP task
in	<i>envv</i>	Environment variables for the spawned PiP task
in	<i>coreno</i>	CPU core number for the PiP task to be bound to. By default, <i>coreno</i> 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, <i>coreno</i> must be bitwise-ORed with <code>PIP_CPUCORE_ABS</code> . If <code>PIP_CPUCORE_ASIS</code> is specified, then the core binding will not take place.
in, out	<i>pipidp</i>	Specify PiP ID of the spawned PiP task. If <code>PIP_PIPID_ANY</code> 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	<i>before</i>	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 <i>hookarg</i> described below.
in	<i>after</i>	This function is called when the PiP task terminates for the cleanup purpose. This function is called with the argument <i>hookarg</i> described below.
in	<i>hookarg</i>	The argument for the <i>before</i> and <i>after</i> function call.

Returns

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

Return values

<i>EPERM</i>	PiP library is not yet initialized
<i>EPERM</i>	PiP task tries to spawn child task
<i>EINVAL</i>	<i>progp</i> is NULL
<i>EINVAL</i>	<i>opts</i> is invalid and/or unacceptable
<i>EINVAL</i>	the value off <i>pipidp</i> is invalid

<i>EINVAL</i>	the coreno is larger than or equal to <code>PIP_CPUCORE_CORENO_MAX</code>
<i>EBUSY</i>	specified PiP ID is already occupied
<i>ENOMEM</i>	not enough memory
<i>ENXIO</i>	<code>dlopen</code> failss

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](#)
[pip_spawn_from_main](#)
[pip_spawn_from_func](#)
[pip_spawn_hook](#)
[pip_task_spawn](#)

3.3 Export/Import Functions

Functions

- int [pip_named_export](#) (void *exp, const char *format,...) `__attribute__((format(printf, 2, 3)))`
export an address of the calling PiP root or a PiP task to the others.
- int int [pip_named_import](#) (int pipid, void **expp, const char *format,...) `__attribute__((format(printf, 3, 4)))`
import the named exported address
- int int int [pip_named_tryimport](#) (int pipid, void **expp, const char *format,...) `__attribute__((format(printf, 3, 4)))`
import the named exported address (non-blocking)
- int int int int [pip_export](#) (void *exp)
export an address
- int [pip_import](#) (int pipid, void **expp)
import exported address of a PiP task

3.3.1 Detailed Description

3.3.1.1 PiP Export and Import

Description

Export and import functions to exchange addresses among tasks

3.3.2 Function Documentation

3.3.2.1 int pip_named_export (void * exp, const char * format, ...)

Name

`pip_named_export`

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 simple `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

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

Returns

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

Return values

<code>EPERM</code>	<code>pip_init</code> is not yet called.
<code>EBUSY</code>	The name is already registered.
<code>ENOMEM</code>	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`
`pip_named_tryimport`
`pip_export`
`pip_import`

3.3.2.2 int int pip_named_import (int *pipid*, void ** *expp*, const char * *format*, ...)

Name

`pip_named_import`

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.

Parameters

in	<i>pipid</i>	The PiP ID to import the exposed address
out	<i>expp</i>	The starting address of the exposed region of the PiP task specified by the <i>pipid</i> .
in	<i>format</i>	a <code>printf</code> format to give the exported address a name

Note

There is a 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

<i>EPERM</i>	<code>pip_init</code> is not yet called.
<i>EINVAL</i>	The specified <i>pipid</i> is invalid
<i>ENOMEM</i>	Not enough memory
<i>ECANCELED</i>	The target task is terminated
<i>EDEADLK</i>	<i>pipid</i> is the calling task and tries to block itself

See Also

[pip_named_export](#)
[pip_named_tryimport](#)
[pip_export](#)
[pip_import](#)

3.3.2.3 `int int int pip_named_tryimport (int pipid, void **expp, const char *format, ...)`

Name

`pip_named_tryimport`

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	<i>pipid</i>	The PiP ID to import the exposed address
out	<i>expp</i>	The starting address of the exposed region of the PiP task specified by the <i>pipid</i> .

<i>in</i>	<i>format</i>	a <code>printf</code> 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

<i>EPERM</i>	<code>pip_init</code> is not yet called.
<i>EINVAL</i>	The specified <code>pipid</code> is invalid
<i>ENOMEM</i>	Not enough memory
<i>ECANCELED</i>	The target task is terminated
<i>EAGAIN</i>	Target is not exported yet

See Also

[pip_named_export](#)
[pip_named_import](#)
[pip_export](#)
[pip_import](#)

3.3.2.4 int int int int pip_export (void * exp)**Name**

`pip_export`

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

<i>in</i>	<i>exp</i>	An addresss
-----------	------------	-------------

Returns

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

Return values

<i>EPERM</i>	PiP library is not initialized yet
--------------	------------------------------------

See Also

[pip_import](#)
[pip_named_export](#)
[pip_named_import](#)
[pip_named_tryimport](#)

3.3.2.5 `int pip_import (int pipid, void ** expp)`

Name

`pip_import`

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	<i>pipid</i>	The PiP ID to import the exported address
out	<i>expp</i>	The exported address

Returns

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

Return values

<i>EPERM</i>	PiP library is not initialized yet
--------------	------------------------------------

See Also

[pip_export](#)
[pip_named_export](#)
[pip_named_import](#)
[pip_named_tryimport](#)

3.4 Waiting for PiP task termination

Functions

- `int pip_wait (int pipid, int *status)`
wait for the termination of a PiP task
- `int pip_trywait (int pipid, int *status)`
wait for the termination of a PiP task in a non-blocking way
- `int pip_wait_any (int *pipid, int *status)`
Wait for the termination of any PiP task.
- `int pip_trywait_any (int *pipid, int *status)`
non-blocking version of [pip_wait_any](#)

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.

3.4.2 Function Documentation

3.4.2.1 `int pip_wait (int pipid, int * status)`

Name

`pip_wait`

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	<i>pipid</i>	PiP ID to wait for.
out	<i>status</i>	Status value of the terminated PiP task

Returns

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

Return values

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

See Also

[pip_exit](#)
[pip_trywait](#)
[pip_wait_any](#)
[pip_trywait_any](#)
[wait\(Linux 2\)](#)

3.4.2.2 `int pip_trywait (int pipid, int * status)`

Name

`pip_trywait`

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	<i>pipid</i>	PiP ID to wait for.
out	<i>status</i>	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

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

See Also

[pip_exit](#)
[pip_wait](#)
[pip_wait_any](#)
[pip_trywait_any](#)
[wait\(Linux 2\)](#)

3.4.2.3 int pip_wait_any (int * *pipid*, int * *status*)

Name

`pip_wait_any`

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	<i>pipid</i>	PiP ID of terminated PiP task.
out	<i>status</i>	Exit status of the terminated PiP task

Returns

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

Return values

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

See Also

[pip_exit](#)
[pip_wait](#)
[pip_trywait](#)
[pip_trywait_any](#)
[wait\(Linux 2\)](#)

3.4.2.4 int pip_trywait_any (int * *pipid*, int * *status*)

Name

`pip_trywait_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	<i>pipid</i>	PiP ID of terminated PiP task.
out	<i>status</i>	Exit status of the terminated PiP task

Returns

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

Return values

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

See Also

[pip_exit](#)
[pip_wait](#)
[pip_trywait](#)
[pip_wait_any](#)
[wait\(Linux 2\)](#)

3.5 PiP Query Functions

Functions

- int [pip_get_pipid](#) (int *pipidp)

- get PiP ID of the calling task*
- int `pip_is_initialized` (void)
Query is PiP library is already initialized.
- int `pip_get_ntasks` (int *ntasksp)
get the maximum number of the PiP tasks
- int `pip_get_mode` (int *modep)
get the PiP execution mode
- const char * `pip_get_mode_str` (void)
get a character string of the current execution mode
- int `pip_get_system_id` (int pipid, pip_id_t *idp)
deliver a process or thread ID defined by the system
- int `pip_isa_root` (void)
check if calling PiP task is a PiP root or not
- int `pip_isa_task` (void)
check if calling PiP task is a PiP task or not
- int `pip_is_threaded` (int *flagp)
check if PiP execution mode is pthread or not
- int `pip_is_shared_fd` (int *flagp)
check if file descriptors are shared or not. This is equivalent with the `pip_is_threaded` function.

3.5.1 Detailed Description

3.5.1.1 PiP Query functions

Description

Query functions for PiP task

3.5.2 Function Documentation

3.5.2.1 int `pip_get_pipid` (int * *pipidp*)

Name

`pip_get_pipid`

Synopsis

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

Parameters

out	<i>pipidp</i>	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

<i>EPERM</i>	PiP library is not initialized yet
--------------	------------------------------------

See Also

[pip_init](#)

3.5.2.2 int pip_is_initialized (void)

Name

pip_is_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.

See Also

[pip_init](#)

3.5.2.3 int pip_get_ntasks (int * ntasksp)

Name

pip_get_ntasks

Synopsis

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

Parameters

out	<i>ntasksp</i>	Maximum number of PiP tasks is returned
-----	----------------	---

Returns

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

Return values

<i>EPERM</i>	PiP library is not yet initialized
--------------	------------------------------------

See Also

[pip_init](#)

3.5.2.4 `int pip_get_mode (int * modep)`

Name

`pip_get_mode`

Synopsis

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

Parameters

<code>out</code>	<code>modep</code>	Returned PiP execution mode
------------------	--------------------	-----------------------------

Returns

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

Return values

<code>EPERM</code>	PiP library is not yet initialized
--------------------	------------------------------------

See Also

[pip_get_mode_str](#)

3.5.2.5 `const char* pip_get_mode_str (void)`

Name

`pip_get_mode_str`

Synopsis

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

Returns

Return the name string of the current execution mode. If PiP library is not initialized yet, then this return NULL.

See Also

[pip_get_mode](#)

3.5.2.6 `int pip_get_system_id (int pipid, pip_id_t * idp)`

Name

`pip_get_system_id`

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 of the PiP execution mode.

Parameters

out	<i>pipid</i>	PiP ID of a target PiP task
out	<i>idp</i>	a pointer to store the ID value

Returns

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

Return values

<i>EPERM</i>	The PiP library is not initialized yet
--------------	--

See Also

[getpid\(Linux 2\)](#)

[pthread_self\(Linux 3\)](#)

3.5.2.7 int pip_isa_root (void)**Name**

`pip_isa_root`

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.

See Also

[pip_init](#)

3.5.2.8 int pip_isa_task (void)**Name**

`pip_isa_task`

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.

See Also

[pip_init](#)

3.5.2.9 `int pip_is_threaded (int * flagp)`

Name

`pip_is_threaded`

Synopsis

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

Parameters

out	<i>flagp</i>	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

<i>EPERM</i>	The PiP library is not initialized yet
--------------	--

See Also

[pip_init](#)

3.5.2.10 `int pip_is_shared_fd (int * flagp)`

Name

`pip_is_shared_fd`

Synopsis

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

Parameters

out	<i>flagp</i>	set to a non-zero value if FDs are shared
-----	--------------	---

Returns

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

Return values

<i>EPERM</i>	The PiP library is not initialized yet
--------------	--

See Also

[pip_init](#)

3.6 PiP task termination

Functions

- void [pip_exit](#) (int status)

- terminate the calling PiP task*
- int [pip_kill_all_tasks](#) (void)
kill all PiP tasks
- void [pip_abort](#) (void)
Kill all PiP tasks and then kill PiP root.
- int [pip_kill](#) (int pipid, int signal)
deliver a signal to PiP task

3.6.1 Detailed Description

3.6.1.1 Terminating PiP task

Description

Terminating PiP task(s)

3.6.2 Function Documentation

3.6.2.1 void [pip_exit](#) (int *status*)

Name

[pip_exit](#)

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

<i>in</i>	<i>status</i>	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](#)
[pip_trywait](#)
[pip_wait_any](#)
[pip_trywait_any](#)
[exit\(Linux 3\)](#)
[pthread_exit\(Linux 3\)](#)

3.6.2.2 int pip_kill_all_tasks (void)

Name

pip_kill_all_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

<i>EPERM</i>	The PiP library is not initialized yet
<i>EPERM</i>	Not called from root

3.6.2.3 void pip_abort (void)

Name

pip_abort

Synopsis

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

3.6.2.4 int pip_kill (int pipid, int signal)

Name

pip_kill

Synopsis

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

Parameters

out	<i>pipid</i>	PiP ID of a target PiP task to deliver the signal
out	<i>signal</i>	signal number to be delivered

Returns

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

Return values

<i>EPERM</i>	PiP library is not yet initialized
<i>EINVAL</i>	An invalid signal number or invalid PiP ID is specified

See Also

`tkill(Luinux 2)`

3.7 PiP Siganling Functions

Functions

- int `pip_sigmask` (int *how*, const sigset_t **sigmask*, sigset_t **oldmask*)
set signal mask of the current PiP task
- int `pip_signal_wait` (int *signal*)
wait for a signal

3.7.1 Detailed Description

3.7.1.1 PiP signaling functions

Description

Signaling functions for PiP task

3.7.2 Function Documentation

3.7.2.1 int `pip_sigmask` (int *how*, const sigset_t * *sigmask*, sigset_t * *oldmask*)

Name

`pip_sigmask`

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	<i>how</i>	see sigprogmask or pthread_sigmask
in	<i>sigmask</i>	signal mask
out	<i>oldmask</i>	old signal mask

Returns

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

Return values

<i>EPERM</i>	PiP library is not yet initialized
<i>EINVAL</i>	An invalid signal number or invalid PiP ID is specified

See Also

sigprocmask(Linux 2)
pthread_sigmask(Linux 3)

3.7.2.2 int pip_signal_wait (int *signal*)

Name

pip_signal_wait

Synopsis

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

Description

This function is agnostic to the PiP execution mode.

Parameters

in	<i>signal</i>	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(Linux 3)
sigsuspend(Linux 2)

3.8 PiP Synchronization Functions

Functions

- int [pip_yield](#) (int flag)
Yield.
- int [pip_barrier_init](#) (pip_barrier_t *barrp, int n)
initialize barrier synchronization structure
- int [pip_barrier_wait](#) (pip_barrier_t *barrp)
wait on barrier synchronization in a busy-wait way
- int [pip_barrier_fin](#) (pip_barrier_t *barrp)
finalize barrier synchronization structure

3.8.1 Detailed Description

3.8.1.1 PiP synchronization functions

Description

Synchronization functions for PiP tasks

3.8.2 Function Documentation

3.8.2.1 `int pip_yield (int flag)`

Name

`pip_yield`

Synopsis

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

Parameters

<i>in</i>	<i>flag</i>	to specify the behavior of yielding. Unused and reserved for BLT/ULP (in PiP-v3)
-----------	-------------	--

Returns

This function always succeeds and returns zero.

See Also

`sched_yield`(Linux 2)
`pthread_yield`(Linux 3)

3.8.2.2 `int pip_barrier_init (pip_barrier_t * barrp, int n)`

Name

`pip_barrier_init`

Synopsis

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

Parameters

<i>in</i>	<i>barrp</i>	pointer to a PiP barrier structure
<i>in</i>	<i>n</i>	number of participants of this barrier synchronization

Returns

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

Return values

<i>EPERM</i>	PiP library is not yet initialized or already finalized
<i>EINAVL</i>	<i>n</i> is invalid

See Also

[pip_barrier_wait](#)
[pip_barrier_fin](#)

3.8.2.3 `int pip_barrier_wait (pip_barrier_t * barrp)`

Name

`pip_barrier_wait`

Synopsis

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

Parameters

<i>in</i>	<i>barrp</i>	pointer to a PiP barrier structure
-----------	--------------	------------------------------------

Returns

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

Return values

<i>EPERM</i>	PiP library is not yet initialized or already finalized
--------------	---

See Also

[pip_barrier_init](#)
[pip_barrier_fin](#)

3.8.2.4 `int pip_barrier_fin (pip_barrier_t * barrp)`

Name

`pip_barrier_fin`

Synopsis

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

Parameters

<i>in</i>	<i>barrp</i>	pointer to a PiP barrier structure
-----------	--------------	------------------------------------

Returns

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

Return values

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

See Also

[pip_barrier_init](#)
[pip_barrier_wait](#)

Index

Export/Import Functions, 21

- pip_export, 24
- pip_import, 24
- pip_named_export, 21
- pip_named_import, 22
- pip_named_tryimport, 23

PiP Initialization/Finalization, 13

- pip_fin, 15
- pip_init, 13

PiP Query Functions, 28

- pip_get_mode, 30
- pip_get_mode_str, 31
- pip_get_ntasks, 30
- pip_get_pipid, 29
- pip_get_system_id, 31
- pip_is_initialized, 30
- pip_is_shared_fd, 33
- pip_is_threaded, 32
- pip_isa_root, 32
- pip_isa_task, 32

PiP Siganling Functions, 36

- pip_sigmask, 36
- pip_signal_wait, 37

PiP Synchronization Functions, 37

- pip_barrier_fin, 39
- pip_barrier_init, 38
- pip_barrier_wait, 39
- pip_yield, 38

PiP task termination, 33

- pip_abort, 35
- pip_exit, 34
- pip_kill, 35
- pip_kill_all_tasks, 34

pip-check, 9

pip-man, 10

pip_abort

- PiP task termination, 35

pip_barrier_fin

- PiP Synchronization Functions, 39

pip_barrier_init

- PiP Synchronization Functions, 38

pip_barrier_wait

- PiP Synchronization Functions, 39

pip_exit

- PiP task termination, 34

pip_export

- Export/Import Functions, 24

pip_fin

- PiP Initialization/Finalization, 15

pip_get_mode

- PiP Query Functions, 30

pip_get_mode_str

- PiP Query Functions, 31

pip_get_ntasks

- PiP Query Functions, 30

pip_get_pipid

- PiP Query Functions, 29

pip_get_system_id

- PiP Query Functions, 31

pip_import

- Export/Import Functions, 24

pip_init

- PiP Initialization/Finalization, 13

pip_is_initialized

- PiP Query Functions, 30

pip_is_shared_fd

- PiP Query Functions, 33

pip_is_threaded

- PiP Query Functions, 32

pip_isa_root

- PiP Query Functions, 32

pip_isa_task

- PiP Query Functions, 32

pip_kill

- PiP task termination, 35

pip_kill_all_tasks

- PiP task termination, 34

pip_named_export

- Export/Import Functions, 21

pip_named_import

- Export/Import Functions, 22

pip_named_tryimport

- Export/Import Functions, 23

pip_sigmask

- PiP Siganling Functions, 36

pip_signal_wait

- PiP Siganling Functions, 37

pip_spawn

- Spawning PiP task, 19

pip_spawn_from_func

- Spawning PiP task, 17

pip_spawn_from_main

- Spawning PiP task, 16

pip_spawn_hook

- Spawning PiP task, 17

pip_task_spawn

- Spawning PiP task, 18

pip_trywait

- Waiting for PiP task termination, [26](#)
- [pip_trywait_any](#)
 - Waiting for PiP task termination, [28](#)
- [pip_wait](#)
 - Waiting for PiP task termination, [26](#)
- [pip_wait_any](#)
 - Waiting for PiP task termination, [27](#)
- [pip_yield](#)
 - PiP Synchronization Functions, [38](#)
- [pip_abort](#), [35](#)
- [pip_barrier_fin](#), [39](#)
- [pip_barrier_init](#), [38](#)
- [pip_barrier_wait](#), [39](#)
- [pip_exit](#), [34](#)
- [pip_export](#), [24](#)
- [pip_fin](#), [15](#)
- [pip_get_mode](#), [31](#)
- [pip_get_mode_str](#), [31](#)
- [pip_get_ntasks](#), [30](#)
- [pip_get_pipid](#), [29](#)
- [pip_get_system_id](#), [31](#)
- [pip_import](#), [25](#)
- [pip_init](#), [13](#)
- [pip_is_initialized](#), [30](#)
- [pip_is_shared_fd](#), [33](#)
- [pip_is_threaded](#), [33](#)
- [pip_isa_root](#), [32](#)
- [pip_isa_task](#), [32](#)
- [pip_kill](#), [35](#)
- [pip_kill_all_tasks](#), [35](#)
- [pip_named_export](#), [21](#)
- [pip_named_import](#), [22](#)
- [pip_named_tryimport](#), [23](#)
- [pip_sigmask](#), [36](#)
- [pip_signal_wait](#), [37](#)
- [pip_spawn](#), [20](#)
- [pip_spawn_from_func](#), [17](#)
- [pip_spawn_from_main](#), [16](#)
- [pip_spawn_hook](#), [17](#)
- [pip_task_spawn](#), [18](#)
- [pip_trywait](#), [26](#)
- [pip_trywait_any](#), [28](#)
- [pip_wait](#), [26](#)
- [pip_wait_any](#), [27](#)
- [pip_yield](#), [38](#)
- [pips](#), [12](#)
- [printpipmode](#), [12](#)
- Spawning PiP task, [16](#)
 - [pip_spawn](#), [19](#)
 - [pip_spawn_from_func](#), [17](#)
 - [pip_spawn_from_main](#), [16](#)
 - [pip_spawn_hook](#), [17](#)
 - [pip_task_spawn](#), [18](#)
- Waiting for PiP task termination, [25](#)
 - [pip_trywait](#), [26](#)
 - [pip_trywait_any](#), [28](#)
 - [pip_wait](#), [26](#)