Processs-in-Process (PiP)

2.0.0

Refernce Manual

September 26, 2020

Generated by Doxygen 1.8.5

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.

Mailing List

pip@ml.riken.jp

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

Commands

- pipcc
- pips
- printpipmode

Functions

- pip_abort
- 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 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
pip-mode

2.2 pip-check

PiP binary checking program.

Synopsis

pip-check [OPTIONS] **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 (default)
-v	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

-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 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 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
printpipmode

2.6 pips

List or kill running PiP tasks.

Synopsis

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

Parameters

-s	Send the specified signal (followed by the "-b" option) to the specified PiP tasks
-k	same as **-s**
-l	List (using the Linux ps command) running PiP tasks specified. This is the default action.
-t	Show running PiP tasks specified by using the top command. Due to the top 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.

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.

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 executionn 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 alomost same way with that of normal Linux process. There are three ways implmented; 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

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 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 ennvironment is set to process:preload. See also the pip_mode(1) command to set the environment variable appropriately and easily.
- **PIP_STACKSZ** Sepcifying the stack size (in bytes). The **KMP_STACKSIZE** and **OMP_STACKSIZE** are also effective. The 't', 'g', 'm', 'k' and 'b' posfix 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 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
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

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

3.2 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 Spawnig 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`	*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
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	*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
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`	*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

3.2.2.4 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	*progp*	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
pip_spawn_from_main
pip_spawn_from_func
pip_spawn_hook
pip_spawn
pip_blt_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`	*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

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
pip_blt_spawn

3.3 Terminating PiP Task

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.

3.3.1 Detailed Description

3.3.1.1 Terminating PiP task

Description

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

3.3.2 Function Documentation

3.3.2.1 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

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)

3.3.2.2 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

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

3.3.2.3 void pip_abort (void)

Name

pip_abort

Synopsis

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

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`	*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)

3.4.2.2 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`	*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)

3.4.2.3 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	*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)

3.4.2.4 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	*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

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

  import the named exported address
- int int int pip_named_tryimport (int pipid, void **expp, const char *format,...) __attribute__((format(printf

  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

3.5.1 Detailed Description

3.5.1.1 PiP Export and Import

Description

Export and import functions to exchange addresses among tasks

3.5.2 Function Documentation

3.5.2.1 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 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

3.5.2.2 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. 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 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

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
pip_named_tryimport
pip_export
pip_import

3.5.2.3 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	*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
pip_named_import
pip_export
pip_import

3.5.2.4 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

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)

3.5.2.5 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	*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)
  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.*

3.6.1 Detailed Description

3.6.1.1 PiP miscellaneous functions

Description

Miscellaneous functions for PiP task (not BLT/ULP)

3.6.2 Function Documentation

3.6.2.1 int pip_get_pipid (int * *pipidp*)

Name

pip_get_pipid

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

3.6.2.2 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.

3.6.2.3 int pip_get_ntasks (int * *ntasksp*)

Name

pip_get_ntasks

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

3.6.2.4 int pip_get_mode (int * *modep*)

Name

pip_get_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)

3.6.2.5 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 note initialized yet, then thiss return `NULL`.

3.6.2.6 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 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

3.6.2.7 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.

3.6.2.8 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.

3.6.2.9 int pip_is_threaded (int * *flagp*)

Name

pip_is_threaded

Synopsis

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

Parameters

out	*flagp*	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

3.6.2.10 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	*flagp*	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)
  deliver a signal to PiP task
- 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

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

3.7.2 Function Documentation

3.7.2.1 int pip_kill (int *pipid,* int *signal*)

Name

pip_kill

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)

3.7.2.2 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	*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

3.7.2.3 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	*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)

  Yield.

4.1.1 Detailed Description

4.1.1.1 Yielding functions

Description

Yielding execution of the calling BLT/ULP

4.1.2 Function Documentation

4.1.2.1 int pip_yield (int *flag*)

Name

pip_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.

Parameters

flag	If **PIP_YIELD_USER**, the calling task is scheduling PiP task(s) then the calling task switch to the next eligible-to-run BLT. If **PIP_YIELD_SYSTEM**, regardless if the calling task is active or inactive, it calls `sched_yield`. If **PIP_YIELD_DEFAULT** or zero, then both **PIP_YIELD_USER** and **PIP_YIELD_SYSTEM** will be effective.

See Also

pip_yield_to(3)

Index