PiP - Process-in-Process

Generated by Doxygen 1.8.5

Wed Jul 1 2020 14:31:51

Contents

Chapter 1

pip_init

the PiP library

Initialize the PiP library

Synopsis:

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

Description:

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

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

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

Parameters

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

Notes:

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

Returns

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

Return values

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

Environment:

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

Bugs:

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

See Also

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

Chapter 2

pip_fin

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)

Chapter 3

pip_spawn_from_main

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

Synopsis:

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

Description:

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

Parameters

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

See Also

pip_task_spawn(3), pip_spawn_from_func(3)

Chapter 4

pip_spawn_from_func

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

Synopsis:

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

Description:

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

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

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

Parameters

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

See Also

pip_task_spawn(3), pip_spawn_from_main(3)

Chapter 5

pip_spawn_hook

Setting invocation hook information

Synopsis:

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

Description:

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

Precondition

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

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

Parameters

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

Note

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

See Also

pip_task_spawn(3)

Chapter 6

pip_task_spawn

Spawning a PiP task

Synopsis:

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

Description:

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

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

Parameters

`in`	*progp*	Program information to spawn as a PiP task
`in`	*coreno*	Core number for the PiP task to be bound to. 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
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.

Bugs:

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

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

See Also

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

Chapter 7

pip_named_export

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

Synopsis:

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

Description:

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

Parameters

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

Returns

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

Return values

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

Note

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

See Also

pip_named_import(3)

Chapter 8

pip_named_import

import the named exported address

Synopsis:

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

Description:

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

Parameters

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

Note

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

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

Returns

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

Return values

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

See Also

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

Chapter 9

pip_named_tryimport

import the named exported address (non-blocking)

Synopsis:

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

Description:

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

Parameters

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

Note

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

Returns

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

Return values

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

See Also

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

Chapter 10

pip_export

export an address

Synopsis:

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

Description:

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

Parameters

in	*exp*	An addresss

Returns

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

Return values

EPERM	PiP library is not initialized yet

See Also

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

Chapter 11

pip_import

import exported address of a PiP task

Synopsis:

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

Description:

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

Parameters

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

Returns

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

Return values

EPERM	PiP library is not initialized yet

See Also

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

Chapter 12

pip_get_pipid

get PiP ID of the calling task

Synopsis:

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

Parameters

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

Returns

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

Return values

EPERM	PiP library is not initialized yet

Chapter 13

pip_is_initialized

Query is PiP library is already initialized

Synopsis:

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

Returns

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

Chapter 14

pip_get_ntasks

get the maximum number of the PiP tasks

Synopsis:

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

Parameters

out	*ntasksp*	Maximum number of PiP tasks is returned

Returns

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

Return values

EPERM	PiP library is not yet initialized

Chapter 15

pip_get_mode

get the PiP execution mode

Synopsis:

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

Parameters

out	*modep*	Returned PiP execution mode

Returns

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

Return values

EPERM	PiP library is not yet initialized

See Also

pip_get_mode_str(3)

Chapter 16

pip_get_mode_str

get a character string of the current execution mode

Synopsis:

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

Returns

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

Chapter 17

pip_exit

terminate the calling PiP task

Synopsis:

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

Description:

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

Parameters

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

Note

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

See Also

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

Chapter 18

pip_wait

wait for the termination of a PiP task

Synopsis:

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

Description:

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

Parameters

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

Returns

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

Return values

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

See Also

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

Chapter 19

pip_trywait

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

Synopsis:

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

Description:

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

Synopsis:

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

Parameters

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

Note

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

Returns

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

Return values

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

See Also

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

Chapter 20

pip_wait_any

Wait for the termination of any PiP task

Synopsis:

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

Description:

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

Parameters

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

Returns

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

Return values

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

See Also

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

Chapter 21

pip_trywait_any

non-blocking version of `pip_wait_any`

Synopsis:

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

Description:

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

Parameters

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

Returns

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

Return values

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

See Also

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

Chapter 22

pip_get_system_id

deliver a process or thread ID defined by the system

Synopsis:

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

Description:

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

Parameters

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

Returns

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

Return values

EPERM	The PiP library is not initialized yet

Chapter 23

pip_isa_root

check if calling PiP task is a PiP root or not

Synopsis:

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

Returns

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

Chapter 24

pip_isa_task

check if calling PiP task is a PiP task or not

Synopsis:

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

Returns

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

Chapter 25

pip_kill_all_tasks

kill all PiP tasks

Synopsis:

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

Note

This function must be called from PiP root.

Returns

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

Return values

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

Chapter 26

pip_abort

kill all PiP tasks and PiP root

Synopsis:

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

Chapter 27

pip_spawn

spawn a PiP task (PiP v1 API and deprecated)

Synopsis:

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

Description:

This function spawns a PiP task.

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*	Core number for the PiP task to be bound to. 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.

Chapter 28

pip_is_threaded

check if PiP execution mode is pthread or not

Synopsis:

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

Parameters

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

Returns

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

Return values

EPERM	The PiP library is not initialized yet

Chapter 29

pip_is_shared_fd

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

Synopsis:

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

Parameters

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

Returns

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

Return values

EPERM	The PiP library is not initialized yet

Chapter 30

pip_decouple

Decouple the curren task from the kernel thread

Parameters

`in`	*task*	specify the scheduling task to schedule the decoupled task (calling this function)

Returns

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

Return values

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

Chapter 31

README

Chapter 32

(This is a Doxygen directive and just ignore)

Process-in-Process (PiP)

Description

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 (fromm v2).

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

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

HTML

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

Getting Started

To 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 pipmodule.c
$ pipcc -Wall -O2 -g -o pipprog pipprog.c
```

To run your PiP programs

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

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 (fromm 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 (fromm v2)

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

List the PiP tasks via the 'ps' command;

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

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

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

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

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

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

Debugging your PiP programs by the pip-gdb command

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

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

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

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

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

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

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

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

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

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

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

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

You can attach all relevant PiP tasks by:

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

(from v2)

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

FAQ

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

Publications

Research papers

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

Presentation Slides

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

Query

Send e-mails to pip@ml.riken.jp

Author

Atsushi Hori Riken Center for Commputational Science (R-CCS) Japan

Chapter 33

Module Index

33.1 Modules

Here is a list of all modules:

Chapter 34

Module Documentation

34.1 pip-exec

run programs as PiP tasks

run programs as PiP tasks

34.1.1 SYNOPSIS

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

34.1.2 DESCRIPTION

Run a program as PiP task(s). Mutiple programs can be specified by separating them with ':'.

-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

34.2 pipcc

C compiler driver for PiP.

C compiler driver for PiP.

34.2.1 SYNOPSIS

pipcc [**pip-options**] [**cc-command-options**]

34.2.2 OPTIONS

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.

34.2.2.1 -piproot

If specified, the compile (and link) is done for PiP root.

34.2.2.2 -piptask

If specified, the compile (and link) is done for PiP task

34.3 pip-mode

Set PiP execution mode.

Set PiP execution mode.

34.3.1 SYNOPSIS

pip-mode [**pipenv-option**] [**commands** ...]

34.3.2 OPTIONS

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.

34.3.2.1 -P

If specified, PiP will run with the 'process' mode

34.3.2.2 -p

If specified, PiP will run with the 'process' mode (preload)

34.3.2.3 -c

If specified, PiP will run with the 'process' mode (pip clone)

34.3.2.4 -t

If specified, PiP will run with the 'thread' mode

34.4 pip-check

PiP binary checking program.

PiP binary checking program.

34.4.1 SYNOPSIS

pipcheck pip-prog [...]

34.5 pip-man

show PiP man page

show PiP man page

34.5.1 SYNOPSIS

pip-man [**MAN_OPTS**] [**PIP_TOPIC**]

34.6 pips

List or kill running PiP tasks.

List or kill running PiP tasks.

34.6.1 SYNOPSIS

pips [**options**] [**pip-command** ...]

34.6.2 OPTIONS

The following options are avilable. If none of them specified, then this sends TERM signal to all running PiP tasks including PiP root.

34.6.2.1 -s \b SIGNAL

Send the specified signal to the specified PiP tasks

34.6.2.2 -k

same as **-s TERM**

34.6.2.3 -l

List (ps command) running PiP tasks specified. This is the default action.

34.6.2.4 --list

same as **-l**

34.6.2.5 -t

Show running PiP tasks specified by using the top command. Due to the top command limitation, only 20 PiP tasks will be shown.

34.6.2.6 --top

same as **-t**

34.7 printpipmode

command to print current PiP mode

command to print current PiP mode

34.7.1 SYNOPSIS

printpipmode

34.7.2 DESCRIPTION

This command prints the current PiP mode setting

34.8 libpip

the PiP library

- int pip_kill (int pipid, int signal)
  deliver a signal to a 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

34.8.1 Detailed Description

the PiP library

34.8.2 Function Documentation

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

deliver a signal to a PiP task

Parameters

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

Note

Only the PiP task can be the target of the signal delivery.
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	PiP library is not yet initialized
EINVAL	An invalid signal number or invalid PiP ID is specified

34.8.2.2 int pip_sigmask (int *how,* const sigset_t * *sigmask,* sigset_t * *oldmask*)

set signal mask of the current PiP task

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**

34.8.2.3 int pip_signal_wait (int *signal*)

wait for a signal

Parameters

in	*signal*	signal to wait

Returns

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

See Also

sigwait, **sigsuspend**

34.9 piplnlibs

command to create symbolic links to the SOLIBs in the patched GLIBC.

command to create symbolic links to the SOLIBs in the patched GLIBC.

34.9.1 SYNOPSIS

piplnlibs [**-rs**]

34.9.2 DESCRIPTION

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

34.9.3 OPTIONS

34.9.3.1 -r

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

34.9.3.2 -s

Silent mode.

34.9.4 ENVIRONMENT

34.9.4.1 PIP_LIBRARY_PATH

Symbolic links to SOLIBs in the directories specified by PIP_LIBRARY_PATH will be also created.

34.9.4.2 LD_LIBRARY_PATH

If PIP_LIBRARY_PATH is not set, LD_LIBRARY_PATH is used instead.

34.10 pip-overview

the PiP library

the PiP library

34.10.1 Overview

PiP is a user-level library which 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 static variables. So, each process runs independently from the other process. If some or all processes agreed, then data own 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 processes. The parent process is called *PiP process* and sub-processes are called *PiP task* since it has the best of the both worlds of processes and pthreads.

PiP root can spawn one or more number of PiP tasks. The executable of the PiP task must be compiled (with the "-fpie" or "-pic" compile option) and linked (with the "-pie" linker option) to be a PIE (Position Independent Executable).

When a PiP root or PiP task is willing to be accessed the its own data by the other(s), firstly a memory region where the data to be accessed are located must be *exported*. Then the exported memory region is *imported* so that the exported and imported data can be accessed. The PiP library supports the functions to export and import the pointers to memory regions.

Unlike shared memory techniques (shared mmap, POSIX-shmem, SYSV-shmem, and XPMEM), PiP allows PiP root and PiP tasks share the entire virtual address space fromm the beginning. So, any pointers can be dereferenced as they are. Additionally there is no need to cast a spell to share a memory region. All users have to do is passing pointers.

34.10.2 Thread

34.10.2.1 Thread

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.

34.10.2.2 Process

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.

34.10.3 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,

- `Pthread`, and
- `Process`.

In the pthread mode, although each PiP task has its own 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, 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" 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 two implementations in the **process** mode; "process:preload" and "process:pipclone" The former one must be with the **LD_PRELOAD** environment variable setting so that the **clone()** system call wrapper can work with. The latter one 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

34.10.4 Limitation

PiP allows PiP root and PiP tasks to share the data, so the function pointer can be passed to the others. However, jumping into the code owned by the other may not work properly for some reasons.

34.10.5 Compile and Link User programs

The PiP root ust be linked with the PiP library and libpthread. The programs able to run as a PiP task must be compiled with the "-fpie" compile option and the "-pie -rdynamic" linker options.

34.10.6 GLIBC issues

The PiP library is implemented at the user-level, i.e. no need of kernel patches nor kernel modules. Due to the novel usage of combining `dlmopn()` GLIBC function and `clone()` systemcall, there are some issues found in the GLIBC. To avoid this issues, PiP users are recommenmded to have the patched GLIBC provided by the PiP development team.

34.10.7 PiP-GDB

The normal gdb debugger only works with the PiP root. PiP-aware GDB (PiP-gdb) is also provided and must be used for debugging PiP tasks. In PiP-gdb, PiP tasks and root can be debugged as GDB inferiors. The current PiP-gdb does not work with the PiP's thread execution mode.

34.10.8 Debug on Exception Signals

If the PIP_GDB_PATH environment is set to the path to PiP-gdb, 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 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 contains GDB commands, then those GDB commands will be executed by the GDB in batch mode.

See Also

pipcc

34.10.9 Author

Atsushi Hori (RIKEN, Japan) ahori@riken.jp

Index