Processs-in-Process (PiP)

2.0.0

Refernce Manual

October 13, 2020

Generated by Doxygen 1.8.5

Contents

1 Proces-in-Process (PiP) Overview **1**

2 PiP Commands **9**

2.1 pipcc 9
2.2 pip-check 9
2.3 pip-exec 10
2.4 pip-man 10
2.5 pip-mode 10
2.6 pips 12
2.7 printpipmode 12

3 PiP Functions **13**

Index **14**

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 owned by a process can be accessed by the other processes. Those processes share the same address space, just like pthreads, but each process has its own variables like the process execution model. Hereinafter, the parent process is called PiP process and a sub-process are called a PiP task.

PiP Versions

Currently there are three PiP library versions:

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

Unfortunately each version has unique ABI and there is no ABI compatibility among them. The functionality of PiP-v1 is almost the same with PiP-v2, however, PiP-v2's API is a subset of the PiP-v3's API. Hereafter **NN** 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;

- Process and
- (P)Thread.

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

When the **PIP_MODE** environment variable set to "()thread" then the PiP library runs in the pthread mode, and if it is set to "process" then it runs in the process mode. There are also three implementations in the process mode; "process:preload," "process: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 is only effective with the PIP-patched glibc library (see below).

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

License

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

Installation

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

Docker image

Download and run the PiP Docker image.

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

Spack

Download spack and do the follwoing;

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

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-NN-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, all packages will be install in "/usr" directory.

Source Code

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

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

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

- `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.

Installation from the source code.

In addition to the above three PiP related packages, there is PiP installing program.

- `PiP-pip` - PiP package installing program

This is the easiest way to install PiP packages from the source code. This program clones all source code from the GITHUB repos, build and install them including PiP documents. Here is the usage of PiP-pip command;

```
$ git clone https://github.com/RIKEN-SysSoft/PiP-pip.git
$ cd PiP-pip
$ ./pip-pip --pip=PIP_VERSION --build=BUILD_DIR --prefix=INSTALL_DIR
```

PiP Documents

The following PiP documents are created by using `Doxygen`.

Man pages

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

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

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

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

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

PDF

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

HTML

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

Getting Started

Compile and link your PiP programs

- pipcc(1) command (since v2)

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

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

Run your PiP programs

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

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

You may write your own PiP programs 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 see how your PiP program is running in realtimme by using the pips(1) command.

List the PiP tasks via the 'ps' command;

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

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

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

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

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

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

Debugging your PiP programs by the pip-gdb command

The following procedure attaches all PiP tasks and PiP root which created those tasks. Each PiP 'processes' is treated as a GDB inferior in PiP-gdb.

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

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

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

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

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

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

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

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

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

(gdb) add-inferior
Added inferior 4
```

```
(gdb) inferior 4
(gdb) attach 1904

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

You can attach all relevant PiP tasks by:

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

(from v2)

If the **PIP_GDB_PATH** environment is set to the path pointing to PiP-gdb executable file, then PiP-gdb is automatically attached when an 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 specified or default signals is delivered, then PiP-gdb will be attached automatically. The PiP-gdb will show backtrace by default. If users specify **PIP_GDB_COMMAND** that a filename containing some GDB commands, then those GDB commands will be executed by PiP-gdb, instead of backtrace, in batch mode. If the **PIP_STOP_ON_START** environment is set (to any value), then the PiP library delivers SIGSTOP to a spawned PiP task which is about to start user program.

Mailing List

pip@ml.riken.jp

Publications

Research papers

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

Atsushi Hori, Balazs Gerofi, and Yuataka Ishikawa. "An Implementation of User-Level Processes using Address Space Sharing," 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), New Orleans, LA, USA, 2020, pp. 976-984, DOI: https://doi.org/10.1109/IPDPSW50202.2020.00161.

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

Commands

- pipcc
- pip-check
- pip-exec
- pip-man

- pip-mode
- pips
- printpipmode

Functions

- pip_abort
- pip_barrier_fin
- pip_barrier_init
- pip_barrier_wait
- pip_exit
- pip_export
- pip_fin
- pip_get_mode
- pip_get_mode_str
- pip_get_ntasks
- pip_get_pipid
- pip_get_system_id
- pip_import
- pip_init
- pip_isa_root
- pip_isa_task
- pip_is_initialized
- pip_is_shared_fd
- pip_is_threaded
- pip_kill
- pip_kill_all_tasks
- pip_named_export
- pip_named_import
- pip_named_tryimport
- pip_sigmask
- pip_signal_wait
- pip_spawn
- pip_spawn_from_func
- pip_spawn_from_main
- pip_spawn_hook

- pip_task_spawn
- pip_trywait
- pip_trywait_any
- pip_wait
- pip_wait_any
- pip_yield

Author

Atsushi Hori

Riken Center for 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

Index

pip-check, 9
pip-man, 10
pips, 12
printpipmode, 12