Processs-in-Process (PiP)

2.0.0

Refernce Manual

September 25, 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

- libpip

Functions

- libpip

Author

Atsushi Hori

Riken Center for Commputational Science (R-CCS)

Japan

Chapter 2

PiP Commands

2.1 libpip

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

- libpip

Functions

- libpip

Author

Atsushi Hori

Riken Center for Commputational Science (R-CCS)

Japan

2.2 md_README

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_INSTALL_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

- libpip

Functions

- libpip

Presentation Slides

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

Author

Atsushi Hori

Riken Center for Commputational Science (R-CCS)

Japan

2.3 md_README-html

2.4 md_README-man

2.5 md_Slides-html

- [HPDC'18] file:/home/ahori/PiP/x86_64/pip-1-1/install/share/slides/HPDC18.pdf
- [ROSS'18] file:/home/ahori/PiP/x86_64/pip-1-1/install/share/slides/HPDC18-ROSS.pdf
- [IPDPS/RADR'20] file:/home/ahori/PiP/x86_64/pip-1-1/install/share/slides/IPDPS-RSADR-2020.pdf

2.6 md_Slides-md

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

Chapter 3

PiP Functions

Chapter 4

BLT/ULP Functions