Great Experiences with PiP (Proecss-in-Process)

Atsushi Hori

March 18, 2023

Scattered wits take a long time in picking up.

Take nothing on its looks; take everything on evidence. Theres no better rule.

I must be taken as I have been made. The success is not mine, the failure is not mine, but the two together make me.

Charles Dickens, *Great Expectations*

Acknowledgment

PVAS (Partitioned Virtual Address Space), created by Akio Shimada, is the predecessor to PiP. He came up with the concept of employing PIE. When Akio and I first met with Pavan Balaji, he already had a keen understanding of the potential of PVAS. He also provided me with some excellent inspiration for PiP. He suggested PiP as the name. Min Si spent a lot of time assisting me with my PiP papers. Excellent articles were written by Kaiming Ouyang to use PiP to enhance MPICH performance. I was able to spend the majority of my time building PiP thanks to Yutaka Ishikawa. Balazs Gerofi also provided me with some insightful feedback on PiP. Noriyuki Soda created the Github actions for testing PiP and really aided me in the development of PiP.

Finally, I want to thank Fusa Hori, my wife, for letting me dedicate my time to writing this book.

Preface

Motivation

I stopped working in April 2022. Since then, I've written this PiP tutorial while also maintaining the PiP library (which includes this document; see https://github.com/procinproc).

PiP offers a distinctive and novel execution model. I've been having a lot of problems developing the PiP library because Linux and the current Glibc don't support the new execution mechanism. I intend for the novel execution model that PiP offers to be significant in computer science in the future, and I will do my best to keep the PiP library up to date.

I am not confident in my ability to adapt PiP to the impending Glibc and Linux because of the fact that Glibc and Linux change much more frequently than PiP does due to manpower constraints. So I made the decision to compose this document to share my thoughts.

This is the reason why this can serve as both an internal document and a tutorial for utilizing PiP.

Expected Readers

In this document, I made every effort to include as many sample programs as I could. They are primarily written in C. A Linux version of the latest PiP library has been tested (CentOS and Redhat, version 7 and 8). Therefore, readers must be conversant with Linux and C programming.

PiP Versions

PiP Version 1 This is the very first release of PiP but it is obsolete now.

PiP Version 2 This is the stable version of PiP.

PiP Version 3 This is a beta version of PiP that uses User-Level Process (ULP) and Bi-Level Thread (BLT). There will be no description of BLT and ULP in this paper because they are not stable at the time of writing.

Other Documents

Not all of the PiP library's features are covered in this document. Consult the PiP reference manual (PDF file at https://github.com/procinproc/PiP/blob/pip-2/doc/latex-inuse/libpip-manpages.pdf), an HTML document, or man pages (man pages and HTML documents will be installed with PiP library) for this purpose.

Sample Programs

Running the sample programs on a Docker environment running on Mac OSX allowed for thorough testing of each program's functionality and the production of all output examples.

Contents

1 PiP Basics **3**
- 1.1 PiP Tasks . . . 3
  - 1.1.1 `pipcc` and `pip-exec` Commands . . . 3
  - 1.1.2 Comparing MPI, OpenMP and PiP . . . 4
  - 1.1.3 Export and Import . . . 6
- 1.2 Spawning PiP Tasks and Waiting Terminations . . . 8
  - 1.2.1 Spawning PiP tasks . . . 8
  - 1.2.2 Waiting for Terminations of PiP tasks . . . 13
  - 1.2.3 Terminating PiP tasks . . . 15
- 1.3 Timing Synchronization among PiP Tasks . . . 16
  - 1.3.1 Barrier Synchronization . . . 16
  - 1.3.2 Using PThread Synchronization . . . 18
  - 1.3.3 `pthread_barrier` . . . 18
  - 1.3.4 `pthread_mutex` . . . 19
- 1.4 PiP Commands . . . 20
  - 1.4.1 `pip-man` . . . 20
  - 1.4.2 `pipcc` and `pipfc` . . . 20
  - 1.4.3 `pip-exec` . . . 21
  - 1.4.4 `pips` . . . 22
  - 1.4.5 `pip-gdb` . . . 23
  - 1.4.6 `pip-mode` and `printpipmode` . . . 23
  - 1.4.7 `libpip.so` . . . 23
- 1.5 Summary . . . 23
- 1.6 Myths on PiP . . . 24

2 PiP Advanced **27**
- 2.1 Rationale . . . 27
- 2.2 Execution Mode . . . 28
  - 2.2.1 Differences Between Two Modes . . . 28
  - 2.2.2 How to Specify Execution Mode . . . 29
- 2.3 Spawning Tasks - Advanced . . . 30
  - 2.3.1 Start Function . . . 30
  - 2.3.2 Stack Size . . . 30

2.3.3 CPU Core Binding . . . 31
2.3.4 File Descriptors and Spawn Hooks . . . 31
2.4 Execution Context . . . 33
2.5 Debugging Support . . . 35
2.5.1 PIP_STOP_ON_START . . . 35
2.5.2 PIP_GDB_SIGNALS . . . 36
2.5.3 PIP_SHOW_MAPS . . . 37
2.5.4 PIP_SHOW_PIPS . . . 37
2.5.5 PIP_GDB_COMMAND . . . 37
2.6 Malloc routines . . . 37
2.7 XPMEM . . . 39

3 PiP Internals 40
3.1 PiP Implementation and design choices . . . 40
3.1.1 Spawning Tasks . . . 40
3.1.2 Calling clone() System Call . . . 41
3.1.3 Execution Mode in Details . . . 42
3.1.4 Name of PiP Tasks . . . 42
3.2 Linux Kernel, Glibc, and Tools Problems . . . 43
3.2.1 Loading a Program . . . 43
3.2.2 Glibc . . . 44
3.2.3 Glibc RPATH Setting . . . 48
3.2.4 Linux . . . 48
3.2.5 Tools . . . 50
3.3 Remaining Issues . . . 51
3.3.1 Retrieving Memory . . . 51

4 PiP Installation 52
4.1 Building from Source Code . . . 52
4.2 **pip-pip command** . . . 53
4.3 Using Spack . . . 54

List of Figures

1.1 Differences of OpenMP, MPI and PiP 6

2.1 Cross-Malloc-Free Issue 38
2.2 Cross-Malloc-Free with Freeing List 38

List of Tables

2.1 Differences between two modes 28
2.2 Mode-Agnostic Functions 29
2.3 Execution Mode Predicates 29
2.4 Possible Signal Names for **PIP_GDB_SIGNALS** 37

3.1 Command Name Setting (1st char.) 42
3.2 Command Name Setting (Second char.) 43
3.3 Glibc functions wrapped by PiP library 45
3.4 Compatibility of Tools 51

Listings

1.1 Hello World (`hello.c`) . . . 3
1.2 Hello World - Compile and Execute . . . 3
1.3 Hello World having a static variable (`hello-var.c`) . . . 4
1.4 Hello World with a static variable - Compile and Execute . . . 4
1.5 Hello World in OpenMP (`hello-var-omp.c`) . . . 5
1.6 Hello World in OpenMP, PiP and MPI - Compile and Execute 5
1.7 Export and Import (`export-import`) . . . 7
1.8 Execution of Export and Import . . . 7
1.9 Spawn (`spawn-root`) . . . 9
1.10 Spawn (`spawn-task`) . . . 9
1.11 Spawn - Execution . . . 10
1.12 Spawn Myself (`spawn-myself`) . . . 10
1.13 Spawn Myself - Execution . . . 11
1.14 Starting from user-defined function (`userfunc`) . . . 11
1.15 Starting from user-defined function - Execution . . . 12
1.16 Starting from main function (`mainfunc`) . . . 12
1.17 Starting from main function - Execution . . . 13
1.18 Waiting for specified PiP task terminations (`wait`) . . . 13
1.19 Waiting for specified PiP task terminations - Execution . . . 14
1.20 Waiting for any PiP task terminations (`waitany`) . . . 14
1.21 Waiting for any PiP task terminations - Execution . . . 15
1.22 PiP Task Termination function (`exit`) . . . 15
1.23 PiP Task Termination - Execution . . . 16
1.24 Barrier Synchronization (`barrier`) . . . 16
1.25 Barrier Synchronization - Execution . . . 17
1.26 Pthread Barrier (`pthread-barrier`) . . . 18
1.27 Pthread Barrier - Execution . . . 19
1.28 Pthread Mutex (`pthread-mutex`) . . . 19
1.29 Pthread Mutex - Execution . . . 20
1.30 `pip-check` - Execution Example . . . 21
1.31 `pip-exec` - Execution Example . . . 22
1.32 `libpip.so` - Execution Example . . . 23
2.1 Before and After Hooks . . . 32
2.2 Before and After Hooks - Execution . . . 33

2.3 Calling a Function of Another Task 33
2.4 Function Call of Another Task - Execution 34
2.5 Stop-on-start Script Example 36
2.6 Stop-on-start Script Example - Execution 36
3.1 `.interp` Section of the `ps` command 43
3.2 Constructors and Destructors 46
3.3 Constructors and Destructors - Execution 47
3.4 A Memory Map Example 48
4.1 Building from Source Code 53
4.2 PiP-pip installation example 53
4.3 Spack installation example 54

Introduction

This document explains the PiP (short for Process-in-Process) library. This library with a little odd name offers a relatively new execution paradigm that combines the finest aspects of multi-process and multi-threaded execution models.

Multiple CPU cores in a CPU die or socket are becoming very common, and a parallel execution environment is essential for maximizing the capability of the many-core architecture. Despite accessing the same physical memory device, a process in the multi-process model, where numerous processes run concurrently on a node, cannot directly access data held by the other processes. Processes typically communicate in order to exchange information. In my opinion, all forms of data copying, whether carried out by hardware or software, are a part of communication. Data copying is energy-, memory-, and time-intensive and should be avoided if possible. What if processes could access other people's data without communicating with them first? In the multi-thread paradigm, static variables are shared across threads, and if threads attempt to update their values, race problems must be prevented. What if static variables were private for each thread?

This is what drives me to develop PiP. As the name *Process-in-Process* might imply, a process can spawn additional processes inside of its own address space. This sounds like the multi-thread execution model, but the name of the *process* in PiP indicates that, unlike the multi-thread approach, each new process has its own static variable set. As a result, while keeping the privatized static variables, the created processes that share the same address space can access data that belongs to the others. On this basis, data copying and communication can be prevented.

In essence, a multi-process model shares nothing, a multi-thread model shares everything, and PiP's execution paradigm allows for everything to be shareable.

Regarding some of my forebears, different implementations exist that offer this particular execution approach. PiP, however, stands out since it is implemented entirely at the user-level and doesn't require a new OS kernel, a patched OS kernel, or new language processing systems.

High performance computing (HPC) has been my area of focus, and I know very little about the other disciplines. HPC applications are all I

can think of. But I think that PiP's usability can be extended to other industries.

Chapter 1

PiP Basics

For those who are unfamiliar, let me first go through the fundamentals of PiP: 1) how to execute a PiP program, 2) how to write a PiP program, and 3) how to use PiP commands. This chapter's explanations don't get into specifics. Consult the other publications (man pages and PDF) or Chapter 2 for further information.

1.1 PiP Tasks

This section will explain how PiP tasks are created simply and how they operate differently from processes (made using MPI) and threads (created using OpenMP).

1.1.1 `pipcc` and `pip-exec` Commands

The first example is the well-known C program "hello world" listed below;

Listing 1.1: Hello World (`hello.c`)

```
#include <stdio.h>
int main() {
  printf( "Hello World\n" );
  return 0;
}
```

As you can see, this program is a perfect match for a standard C program. If the **pipcc** command was used to compile the program, it can be run as a standard C program or as a PiP task by using the **pip-exec** command.

Listing 1.2: Hello World - Compile and Execute

```
$ pipcc --silent hello.c -o hello
$ ./hello
Hello World
$ pip-exec ./hello
```

```
Hello World
$
```

A true C compiler can be called with the proper options, such as `-I`, `-L`, and others, using the **pipcc** command, which is written as a shell script. You will see the options available when the **pipcc** script calls the backend C/C++ compiler if the `silent` option is omitted.

In this case, **pip-exec** is being used to run an executable file as PiP tasks rather than a standard Linux process. The hello program does not operate differently in the process and PiP task in this case. In the following section, we'll talk about this issue.

1.1.2 Comparing MPI, OpenMP and PiP

We slightly alter the "hello world" software as follows to clarify the distinction between the Linux process and PiP task;

Listing 1.3: Hello World having a static variable (`hello-var.c`)

```
#include <stdio.h>
int x;
int main() {
  printf( "Hello World (&x:%p)\n", &x );
  return 0;
}
```

Now, the address of the static variable `x` in the "Hello World" program is printed out along with the message "Hello World." The number of PiP jobs to be created and run concurrently can be set using the **pip-exec** command option. In the execution example that follows, the number three (3) is supplied. The same `a.out` execution using MPI's output is also included. It should be noted that the "Hello World" program runs in parallel with **pip-exec** and `mpiexec`.

Listing 1.4: Hello World with a static variable - Compile and Execute

```
$ pipcc --silent hello-var.c -o hello-var
$ ./hello-var
Hello World (&x:0x555555601030)
$ pip-exec -n 3 ./hello-var
Hello World (&x:0x7ffff67d9030)
Hello World (&x:0x7ffff48db030)
Hello World (&x:0x7fffee92c030)
$ mpiexec -n 3 ./hello-var
Hello World (&x:0x555555601030)
Hello World (&x:0x555555601030)
Hello World (&x:0x555555601030)
$
```

The variable `x` is found at the address `0x5555556010301` according to the first execution of a.out. With MPI execution, this circumstance is same[1]. The variable `x` is however executed at various places for each PiP activity. This is due to MPI jobs not sharing the same address space as PiP tasks.

Readers who are curious in the distinction between PiP and OpenMP may observe that threads also share the same address space. The "Hello World" program with a static variable is shown in the example below written in OpenMP.

Listing 1.5: Hello World in OpenMP (`hello-var-omp.c`)

```
#include <stdio.h>
int x;
int main() {
  #pragma omp parallel
  printf( "Hello World (&x:%p)\n", &x );
  return 0;
}
```

The output of program 1.5's execution is displayed below. The addresses of variable `x` in this case are the same for MPI and OpenMP executions. The variable's addresses with PiP execution, however, are different pairings.

Listing 1.6: Hello World in OpenMP, PiP and MPI - Compile and Execute

```
$ pipcc --silent -fopenmp hello-var-omp.c -o hello-var-omp
$ export OMP_NUM_THREADS=2
$ ./hello-var-omp
Hello World (&x:0x555555601038)
Hello World (&x:0x555555601038)
$ pip-exec -n 2 ./hello-var-omp
Hello World (&x:0x7ffff67d9038)
Hello World (&x:0x7ffff67d9038)
Hello World (&x:0x7fffefde3038)
Hello World (&x:0x7fffefde3038)
$ mpiexec -n 2 ./hello-var-omp
Hello World (&x:0x555555601038)
Hello World (&x:0x555555601038)
Hello World (&x:0x555555601038)
Hello World (&x:0x555555601038)
$
```

These variations are explained in Figure 1.1. The variable `x` is shared by all of the OpenMP threads, and they all use the same address space. Each MPI process in an MPI environment has its own address space, and two (2) threads can execute in each address space while sharing a variable in an MPI process. However, each PiP task has its own variables, thus threads

[1]For simplicity, we disabled ASLR (Address Space Layout Randomization) in this example.

0 and 1 only share variables within the same PiP task; they do not share variables inside any other PiP tasks. In PiP, all PiP tasks share the same address space.

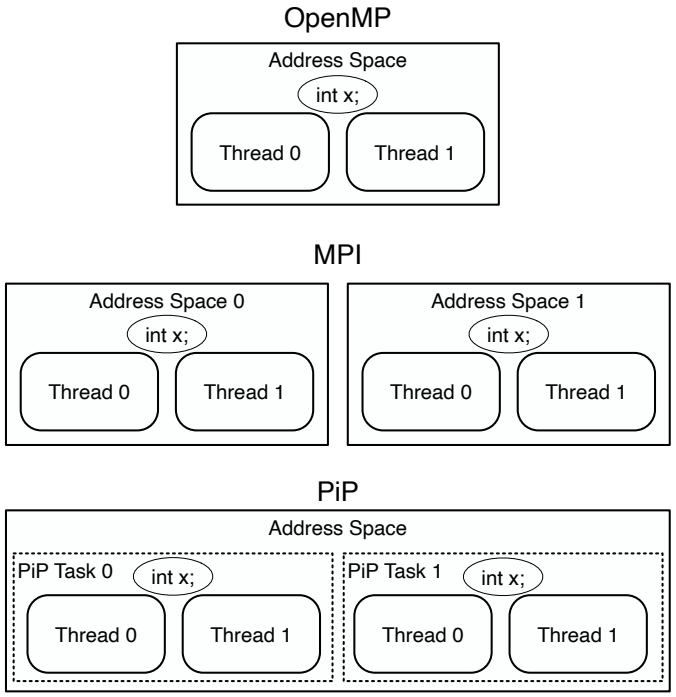

Figure 1.1: Differences of OpenMP, MPI and PiP

Static variables are associated to an address space in the traditional process model and thread model. Because of this, each process has its own static variables, which are shared by all threads using the same address space. Each PiP task is guaranteed to have its own static variable set under the PiP execution model, decoupling from the address space while maintaining address space sharing. *variable privatization* is what this is.

It is simple to share information among PiP tasks while maintaining the independence of each PiP task's execution thanks to the nature of PiP, which includes privatized variables and sharing an address space. The "Hello World" program has so far been proved to be able to execute as PiP tasks concurrently, although this program is quite basic and there is no information exchange between PiP tasks. We'll demonstrate how information can be shared between PiP processes in the section after this one.

1.1.3 Export and Import

If the address of the information to be exchanged is known, sharing an address space allows data owned by a PiP task to be accessed. The address

of the shared data can be broadcast by a PiP task, and the other PiP task(s) can obtain the published address.

To start, each PiP task has a **PIPID** that helps it stand out from the rest. By giving the **PIPID** of the exporting PiP task, other PiP tasks that share the same address space can import the exported address.

Listing 1.7: Export and Import (`export-import`)

```
#include <pip/pip.h>
#include <stdlib.h>
int x;
int main( int argc, char **argv ) {
  int pipid, *xp;
  pip_get_pipid( &pipid );
  if( pipid == 0 ) {
    x = strtol( argv[1], NULL, 10 );;
    pip_named_export( &x, "export" );
  } else {
    pip_named_import( 0, (void**) &xp, "export" );
    printf( "%d: %d\n", pipid, *xp );
  }
  return 0;
}
```

In this program, after adjusting the value of `argv[1]`, a PiP task with a PIPID of zero (zero) exports the address of the variable `x` by using **pip_named_export()**. By using the **pip_named_import()** function, the remaining PiP tasks import the address that PiP task 0 exported. An outcome of this program's execution is shown below. The other PiP tasks can view the value that PiP task 0 exported, as demonstrated.

Listing 1.8: Execution of Export and Import

```
$ pip-exec -n 4 ./export-import 1234
1: 1234
2: 1234
3: 1234
$ pip-exec -n 4 ./export-import 18526
1: 18526
2: 18526
3: 18526
$
```

The address with the specified name is published by the **pip_named_export()** function. The **pip_named_import()** function blocking-waits for the specified PiP job by **PIPID** to reach the defined address. To avoid a race condition, it is not permitted to export an address with the same name more than once for the purpose of updating the address.

The PiP library's functions almost always return an integer value as an error code. A return code of zero (zero) denotes success. This error code

is identical to those that Linux defines. Due to simplicity and clarity, the returned code is not tested in the examples presented thus far and moving forward.

It is forbidden in MPI to access the data that is held by other processes running on the same node[2]. In MPI, communication is the sole permitted method. Communication fundamentally entails copying data in some way (done by software or hardware). Data copying consumes memory, power, and time.

1.2 Spawning PiP Tasks and Waiting Terminations

The **pip-exec** command starts PiP tasks. (PiP) root process is the name of the process that creates PiP tasks. The **pip-exec** process is a root process. The root process's address space is used to map and execute PiP tasks that it has spawned. This chapter will describe how to spawn PiP tasks.

1.2.1 Spawning PiP tasks

Spawning a program as PiP tasks

Listing 1.9 is an example of a PiP root program. It spawns N PiP tasks, where N is specified by the first parameter of the program. The **pip_init()** function must be called to initialize the PiP library before calling any other PiP functions, although there some exceptions to this. The first argument is output returning **PIPID** of the calling task. In this case, **PIP_PIPID_ROOT** is returned, since the function is called by the root. The second input arguments is to specify the maximum number of spawning PiP tasks. The other arguments will be explained in Chapter 2.

The **pip_spawn()** function is called after then. The first and second arguments are the same with the Linux's `execve()` function; the first is to specify the executable file to be executed and the second argument is to specify the parameters executing the program. The third is to specify environment variables. When it is `NULL`, then value of the Glibc global variable `environ` is taken. Th fourth argument is to specify the CPU core number to bind the spawned PiP task and which CPU core. In this example, the value of **PIP_CPUCORE_ASIS** means that the (CPU) core-bind should be the same with the one when calling **pip_spawn()**. The fifth is an input and output argument, and you can specify a **PIPID** or set to the value

[2]Strictly speaking, some MPI implementations based on the thread model may allow this. Major MPI implementation, such MPICH, Open MPI, and many other MPI implementations provided by vendors are based on the process model, and there is no way to access data owned by the other MPI process.

of **PIP_PIPID_ANY** so that PiP library can choose any. After calling **pip_spawn()**, the argument returns the actual **PIPID**.

The **pip_wait()** is to wait the termination of the spawned task. Its first argument is to specify the **PIPID** of the terminating task. The second parameter, although `NULL` is set in this example, is the same with the Linux's `wait()` function, returning the terminating status of a task.

The **pip_fin()** function works as the opposite of **pip_init()**, finalizing PiP library and freeing allocated resources. After calling **pip_fin()**, most PiP library functions return an error code (`EPERM`).

Listing 1.9: Spawn (`spawn-root`)

```
#include <pip/pip.h>
#include <stdlib.h>
int main( int argc, char **argv ) {
  int pipid, ntasks, pipid_task, i;
  ntasks = strtol( argv[1], NULL, 10 );
  pip_init( &pipid, &ntasks, NULL, 0 );
  for( i=0; i<ntasks; i++ ) {
    pipid_task = PIP_PIPID_ANY;
    pip_spawn( argv[2], &argv[2], NULL,
               PIP_CPUCORE_ASIS, &pipid_task,
               NULL, NULL, NULL );
    pip_wait( pipid_task, NULL );
    printf( "PiP task (PIPID:%d) done\n",
            pipid_task );
  }
  pip_fin();
  return 0;
}
```

Listing 1.10 is very similar to the "Hello World" program in the previous section. The major difference here is calling the **pip_init()** function. The **pip_init()** may look strange because this function behaves differently depending on if it is called from a PiP root or PiP task. Unlike root, this function call is optional in the PiP task program. By calling this, you can get **PIPID** and the number of maximum PiP tasks which are specified by the root. Linsting 1.11 shows an example of the execution of Listing 1.9 and 1.10.

Listing 1.10: Spawn (`spawn-task`)

```
#include <pip/pip.h>
int main( int argc, char **argv ) {
  int pipid, ntasks;
  pip_init( &pipid, &ntasks, NULL, 0 );
  printf( "\"%s\" from PIPID:%d/%d\n",
          argv[1], pipid, ntasks );
  pip_fin();
```

```
  return 0;
}
```

Listing 1.11: Spawn - Execution

```
$ ./spawn-root 4 ./spawn-task "What's up?"
"What's up?" from PIPID:0/4
"What's up?" from PIPID:1/4
"What's up?" from PIPID:2/4
"What's up?" from PIPID:3/4
PiP task (PIPID:0) done
PiP task (PIPID:1) done
PiP task (PIPID:2) done
PiP task (PIPID:3) done
$
```

Spawning myself

A program can be either the PiP task or the PiP root. Combining the programs from Listings 1.12 and 1.10 is demonstrated in Listing 1.12 as an example. We hope you can comprehend **pip_init()**'s peculiar behavior. The PiP root process functions similarly to a PiP task. It has a unique **PIPID** called the **PIP_PIPID_ROOT**. Listing 1.13 provides an illustration of this execution.

Listing 1.12: Spawn Myself (`spawn-myself`)

```
#include <pip/pip.h>
#include <stdlib.h>
int main( int argc, char **argv ) {
  int pipid, ntasks, pipid_task, i;
  ntasks = strtol( argv[1], NULL, 10 );
  pip_init( &pipid, &ntasks, NULL, 0 );
  if( pipid == PIP_PIPID_ROOT ) {
    /* PiP root */
    for( i=0; i<ntasks; i++ ) {
      pipid_task = i;
      pip_spawn( argv[0], argv, NULL,
                 PIP_CPUCORE_ASIS, &pipid_task,
                 NULL, NULL, NULL );
      pip_wait( pipid_task, NULL );
      printf( "PiP task (PIPID:%d) done\n",
              pipid_task );
    }
  } else {
    /* PiP task */
    printf( "\"%s\" from PIPID:%d/%d\n",
            argv[2], pipid, ntasks );
  }
```

```
  pip_fin();
  return 0;
}
```

Listing 1.13: Spawn Myself - Execution

```
$ ./spawn-myself 4 "Learning PiP."
"Learning PiP." from PIPID:0/4
"Learning PiP." from PIPID:1/4
"Learning PiP." from PIPID:2/4
"Learning PiP." from PIPID:3/4
PiP task (PIPID:0) done
PiP task (PIPID:1) done
PiP task (PIPID:2) done
PiP task (PIPID:3) done
$
```

Starting from other than `main`

In the preceding instances, PiP tasks start from the `main()` function. PiP enables tasks to launch user-defined functions other than the `main()`. Use the **pip_task_spawn()** function in this situation rather than executing **pip_spawn()**.

Listing 1.14: Starting from user-defined function (`userfunc`)

```
#include <pip/pip.h>
#include <stdlib.h>
int user_func( void *arg ) {
  char *msg = (char*) arg;
  int pipid, ntasks;
  pip_get_pipid( &pipid );
  pip_get_ntasks( &ntasks );
  printf( "USER-FUNC: \"%s\" from PIPID:%d/%d\n",
          msg, pipid, ntasks );
  return 0;
}

int main( int argc, char **argv ) {
  int pipid, ntasks, pipid_task, i;
  ntasks = strtol( argv[1], NULL, 10 );
  pip_init( &pipid, &ntasks, NULL, 0 );
  if( pipid == PIP_PIPID_ROOT ) {
    pip_spawn_program_t prog;
    pip_spawn_from_func( &prog,
                         argv[0],      /* exec file */
                         "user_func", /* func name */
                         (void*) argv[2], /* arg    */
                         NULL,         /* environ   */
```

```
                          NULL );/* explained later */
    for( i=0; i<ntasks; i++ ) {
      pipid_task = i;
      pip_task_spawn( &prog, PIP_CPUCORE_ASIS, 0,
                      &pipid_task, NULL );
      pip_wait( pipid_task, NULL );
    }
  } else {
    /* NEVER REACH HERE */
    printf( "MAIN: \"%s\" from PIPID:%d/%d\n",
            argv[2], pipid, ntasks );
  }
  pip_fin();
  return 0;
}
```

The software for this example is listed in Listing 1.14. The **pip_spawn_program_t** structure is defined to minimize the number of arguments needed to spawn a PiP process. All necessary data for starting a program, such as the function name and path to the executable file, are stored in this structure. The **pip_spawn_from_func()** function is also defined to set this information in order to mask the structure's specifics. The user-defined function must take a single argument `(void*)` and return an integer value that is identical to the `main()` function's return value.

Listing 1.15: Starting from user-defined function - Execution

```
$ ./userfunc 4 "Calling user_func"
USER-FUNC: "Calling user_func" from PIPID:0/4
USER-FUNC: "Calling user_func" from PIPID:1/4
USER-FUNC: "Calling user_func" from PIPID:2/4
USER-FUNC: "Calling user_func" from PIPID:3/4
$
```

The **pip_spawn()** was initially released (from version 1). After that, I found users could launch PiP tasks other than the `main()`, and the **pip_task_spawn()** function had been introduced (from version 2 or later). When beginning from the `main()` function, the **pip_spawn_program_t** structure must be set by calling the **pip_spawn_from_main()** function. Listing 1.16 is a program that updates Listing 1.12 to use the **pip_task_spawn()** and **pip_spawn_from_main()** functions.

Listing 1.16: Starting from main function (`mainfunc`)

```
#include <pip/pip.h>
#include <stdlib.h>
int main( int argc, char **argv ) {
  int pipid, ntasks, pipid_task, i;
  ntasks = strtol( argv[1], NULL, 10 );
  pip_init( &pipid, &ntasks, NULL, 0 );
```

```
  if( pipid == PIP_PIPID_ROOT ) {
    pip_spawn_program_t prog;
    pip_spawn_from_main( &prog,
                         argv[0],      /* exec file */
                         argv,         /* argv      */
                         NULL,         /* environ   */
                         NULL );/* explained later */
    for( i=0; i<ntasks; i++ ) {
      pipid_task = i;
      pip_task_spawn( &prog, PIP_CPUCORE_ASIS, 0,
                      &pipid_task, NULL );
      pip_wait( pipid_task, NULL );
    }
  } else {
    printf( "MAIN: \"%s\" from PIPID:%d/%d\n",
            argv[2], pipid, ntasks );
  }
  pip_fin();
  return 0;
}
```

Listing 1.17: Starting from main function - Execution

```
$ ./mainfunc 4 "Calling main"
MAIN: "Calling main" from PIPID:0/4
MAIN: "Calling main" from PIPID:1/4
MAIN: "Calling main" from PIPID:2/4
MAIN: "Calling main" from PIPID:3/4
$
```

1.2.2 Waiting for Terminations of PiP tasks

As readers may have already noted, the **pip_wait()** function is used to watch for PiP tasks that have been started to finish. The **pip_wait()** function functions similarly to the `wait()` function in Linux. Linux's `wait()` function frequently cooperates with PiP jobs, however there is one instance where it does not. Therefore, it is advised that users use the **pip_wait()** function. Similar to the wait() function in Linux, the argument of **pip_wait()** is a pointer to an integer variable. The Linux `WIFEXITED`, `WIFSIGNALED`, `WEXITSTATUS`, `WIFSIGNALED`, and `WTERMISIG` macros can be used to inspect the returned integer.

Listing 1.18: Waiting for specified PiP task terminations (`wait`)

```
#include <pip/pip.h>
#include <stdlib.h>
int main( int argc, char **argv ) {
  int pipid, ntasks, pipid_task, i, exitval = 0;
```

```
  ntasks = strtol( argv[1], NULL, 10 );
  pip_init( &pipid, &ntasks, NULL, 0 );
  if( pipid == PIP_PIPID_ROOT ) {
    for( i=0; i<ntasks; i++ ) {
      int status;
      pipid_task = i;
      pip_spawn( argv[0], argv, NULL,
                 PIP_CPUCORE_ASIS, &pipid_task,
                 NULL, NULL, NULL );
      pip_wait( pipid_task, &status );
      printf( "PiP task (PIPID:%d) done: %d\n",
              pipid_task, WEXITSTATUS(status) );
    }
  } else {
    exitval = pipid;
  }
  pip_fin();
  return exitval;
}
```

Listing 1.19: Waiting for specified PiP task terminations - Execution

```
$ ./wait 4
PiP task (PIPID:0) done: 0
PiP task (PIPID:1) done: 1
PiP task (PIPID:2) done: 2
PiP task (PIPID:3) done: 3
$
```

pip_wait() waits for the PiP task termination specified by **PIPID**. **pip_wait_any()** function can wait for any PiP tasks and **PIPID** and exit status are returned when terminated (See Listing 1.20 and 1.21). **pip_trywait()** and **pip_trywait_any()** are the non-blocking versions of **pip_wait()** and **pip_wait_any()**, respectively.

Listing 1.20: Waiting for any PiP task terminations (`waitany`)

```
#include <pip/pip.h>
#include <stdlib.h>
int main( int argc, char **argv ) {
  int pipid, ntasks, pipid_task, i, exitval = 0;
  ntasks = strtol( argv[1], NULL, 10 );
  pip_init( &pipid, &ntasks, NULL, 0 );
  if( pipid == PIP_PIPID_ROOT ) {
    int status;
    for( i=0; i<ntasks; i++ ) {
      pipid_task = i;
      pip_spawn( argv[0], argv, NULL,
                 PIP_CPUCORE_ASIS, &pipid_task,
                 NULL, NULL, NULL );
```

```
    }
    for( i=0; i<ntasks; i++ ) {
      pip_wait_any( &pipid_task, &status );
      printf( "PiP task (PIPID:%d) done: %d\n",
              pipid_task, WEXITSTATUS(status) );
    }
  } else {
    exitval = pipid;
  }
  pip_fin();
  return exitval;
}
```

Listing 1.21: Waiting for any PiP task terminations - Execution

```
$ ./waitany 4
PiP task (PIPID:0) done: 0
PiP task (PIPID:1) done: 1
PiP task (PIPID:2) done: 2
PiP task (PIPID:3) done: 3
$
```

1.2.3 Terminating PiP tasks

By using the **pip_exit()** function, PiP tasks and root can stop running. Comparable to Linux's `exit()` function is this function. As stated above, it is advised to use **pip_exit()** rather than `exit()` since, while the Linux `exit()` function typically works, there are some instances where it does not. Listing 1.22 and 1.23 provide a working example of **pip_exit()**.

Listing 1.22: PiP Task Termination function (`exit`)

```
#include <pip/pip.h>
#include <stdlib.h>
int main( int argc, char **argv ) {
  int pipid, ntasks, pipid_task, i, exitval = 0;
  ntasks = strtol( argv[1], NULL, 10 );
  pip_init( &pipid, &ntasks, NULL, 0 );
  if( pipid == PIP_PIPID_ROOT ) {
    for( i=0; i<ntasks; i++ ) {
      int status;
      pipid_task = i;
      pip_spawn( argv[0], argv, NULL,
                 PIP_CPUCORE_ASIS, &pipid_task,
                 NULL, NULL, NULL );
      pip_wait( pipid_task, &status );
      printf( "PiP task (PIPID:%d) done: %d\n",
              pipid_task, WEXITSTATUS(status) );
    }
```

```
    pip_exit( 100 );
    /* NEVER REACH HERE */
  } else {
    exitval = pipid * 10;
    pip_exit( exitval );
    /* NEVER REACH HERE */
  }
}
```

Listing 1.23: PiP Task Termination - Execution

```
$ ./exit 4; echo $?
PiP task (PIPID:0) done: 0
PiP task (PIPID:1) done: 10
PiP task (PIPID:2) done: 20
PiP task (PIPID:3) done: 30
100
$
```

1.3 Timing Synchronization among PiP Tasks

This section will go over how PiP tasks' timing synchronization works.

1.3.1 Barrier Synchronization

The PiP library only supports barrier synchronization as a synchronization mechanism at this time. Barrier synchronization in PiP uses an API that was taken from the PThread library. PiP has three functions that are equivalent to `pthread_barrier_init()`, `pthread_barrier_wait()`, and `pthread_barrier_destroy()`, respectively: **pip_barrier_init()**, **pip_barrier_wait()**, and **pip_barrier_fin()**.

Listing 1.24: Barrier Synchronization (`barrier`)

```
#include <pip/pip.h>
#include <stdlib.h>
pip_barrier_t barr;
int main( int argc, char **argv ) {
  int pipid, ntasks, pipid_task, i;
  pip_barrier_t *barrp = &barr;
  ntasks = strtol( argv[1], NULL, 10 );
  pip_init( &pipid, &ntasks, (void**) &barrp, 0 );
  if( pipid == PIP_PIPID_ROOT ) {
    pip_barrier_init( barrp, ntasks );
    for( i=0; i<ntasks; i++ ) {
      pipid_task = i;
      pip_spawn( argv[0], argv, NULL,
                 PIP_CPUCORE_ASIS, &pipid_task,
```

```
                NULL, NULL, NULL );
    }
    for( i=0; i<ntasks; i++ ) {
      pip_wait_any( &pipid_task, NULL );
      printf( "PiP task (PIPID:%d) done\n", pipid_task );
    }
    pip_barrier_fin( barrp );
  } else {
    if( argv[2] == NULL ) {
      pip_barrier_wait( barrp );
    }
    printf( "PIPID:%d %f [S]\n", pipid, pip_gettime() );
  }
  return 0;
}
```

Listing 1.24 gives the third argument to the **pip_init()** function a non-NULL value this time. This is an additional method of exporting a pointer to spawnd PiP jobs from the root. Children in this example are given the address of the **pip_barrier_t** static variable so they can synchronize by using **pip_barrier_wait()**.

To make the impact of the barrier synchronization more clear, the synchronization only occurs when the second parameter of the program execution is left blank; in this case, PiP tasks display the **pip_gettime()** return values. The **pip_gettime()** function returns the current value of the `gettimeofday()` function in double format with a second unit.

Listing 1.25 displays an example of this application being run. The barrier synchronization is absent in the initial run, and the values returned by `gettimeofday()` exhibit significant variation. A lesser difference can be detected in the second run, where the barrier synchronization occurs.

Listing 1.25: Barrier Synchronization - Execution

```
$ ./barrier 4 NOBARRIER
PIPID:0 1661152530.820478 [S]
PIPID:1 1661152530.847696 [S]
PIPID:2 1661152530.878638 [S]
PIPID:3 1661152530.902075 [S]
PiP task (PIPID:0) done
PiP task (PIPID:1) done
PiP task (PIPID:2) done
PiP task (PIPID:3) done
$ ./barrier 4
PIPID:3 1661152531.178360 [S]
PIPID:0 1661152531.178426 [S]
PIPID:1 1661152531.178457 [S]
PIPID:2 1661152531.179018 [S]
PiP task (PIPID:3) done
PiP task (PIPID:0) done
```

```
PiP task (PIPID:1) done
PiP task (PIPID:2) done
$
```

1.3.2 Using PThread Synchronization

The PThread library's synchronization features for PiP tasks are likewise available to users. This is due to the fact that PiP tasks and threads both use the same address space.

1.3.3 `pthread_barrier`

The PThread's barrier functions can also be used to provide the same barrier synchronization. Listing 1.26 shows the program's simple substitution of the PThread's barrier functions for the PiP's barrier functions.

Listing 1.26: Pthread Barrier (`pthread-barrier`)

```
#include <pip/pip.h>
#include <stdlib.h>
#include <pthread.h>
pthread_barrier_t barr;
int main( int argc, char **argv ) {
  int pipid, ntasks, pipid_task, i;
  pthread_barrier_t *barrp = &barr;
  ntasks = strtol( argv[1], NULL, 10 );
  pip_init( &pipid, &ntasks, (void**) &barrp, 0 );
  if( pipid == PIP_PIPID_ROOT ) {
    pthread_barrier_init( barrp, NULL, ntasks );
    for( i=0; i<ntasks; i++ ) {
      pipid_task = i;
      pip_spawn( argv[0], argv, NULL,
                 PIP_CPUCORE_ASIS, &pipid_task,
                 NULL, NULL, NULL );
    }
    for( i=0; i<ntasks; i++ ) {
      pip_wait_any( &pipid_task, NULL );
      printf( "PiP task (PIPID:%d) done\n", pipid_task );
    }
    pthread_barrier_destroy( barrp );
  } else {
    if( argv[2] == NULL ) {
      pthread_barrier_wait( barrp );
    }
    printf( "PIPID:%d %f [S]\n", pipid, pip_gettime() );
  }
  return 0;
}
```

Listing 1.27: Pthread Barrier - Execution

```
$ ./pthread-barrier 4 NOBARRIER
PIPID:0 1661152533.381032 [S]
PIPID:1 1661152533.410988 [S]
PIPID:2 1661152533.451032 [S]
PIPID:3 1661152533.566188 [S]
PiP task (PIPID:0) done
PiP task (PIPID:1) done
PiP task (PIPID:2) done
PiP task (PIPID:3) done
$ ./pthread-barrier 4
PIPID:3 1661152534.127914 [S]
PIPID:1 1661152534.127852 [S]
PIPID:0 1661152534.127925 [S]
PIPID:2 1661152534.128041 [S]
PiP task (PIPID:1) done
PiP task (PIPID:3) done
PiP task (PIPID:0) done
PiP task (PIPID:2) done
$
```

1.3.4 pthread_mutex

Similarly, `pthread_mutex()` also works with PiP.

Listing 1.28: Pthread Mutex (`pthread-mutex`)

```
#include <pip/pip.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#define NITERS          (1000)
typedef struct sync_tasks {
  pthread_barrier_t     barr;
  pthread_mutex_t       mutex;
  int                   count;
} sync_t;
sync_t  sync_tasks;
int lock;
void increment( sync_t *syncp ) {
  int tmp;
  if( lock ) pthread_mutex_lock( &syncp->mutex );
  tmp = syncp->count;
  usleep( 10 );
  syncp->count = tmp + 1;
  if( lock ) pthread_mutex_unlock( &syncp->mutex );
}
int main( int argc, char **argv ) {
  int pipid, ntasks, i;
```

```
  sync_t *syncp;
  pip_get_pipid( &pipid );
  pip_get_ntasks( &ntasks );
  lock = ( argc == 1 );
  if( pipid == 0 ) {
    syncp = &sync_tasks;
    pthread_barrier_init( &syncp->barr, NULL, ntasks );
    pthread_mutex_init( &syncp->mutex, NULL );
    syncp->count = 0;
    pip_named_export( syncp, "sync" );
  } else {
    pip_named_import( 0, (void**) &syncp, "sync" );
  }
  pthread_barrier_wait( &syncp->barr );
  for( i=0; i<NITERS; i++ ) increment( syncp );
  pthread_barrier_wait( &syncp->barr );
  if( pipid == 0 ) {
    printf( "count=%d (%d*%d)\n", syncp->count,
            ntasks, NITERS );
  }
  return 0;
}
```

Listing 1.29: Pthread Mutex - Execution

```
$ pip-exec -n 10 ./pthread-mutex
count=10000 (10*1000)
$ pip-exec -n 10 ./pthread-mutex NOLOCK
count=992 (10*1000)
$
```

1.4 PiP Commands

The PiP package's PiP commands are covered in this section. Some of these have already been demonstrated but just briefly discussed. The specifics of the PiP commands will be described in this section.

1.4.1 `pip-man`

The PiP man pages are displayed by this command. Although using this command does not require users to be concerned with the `MAN_PATH`, it does execute the Linux `man` command with the `MAN_PATH` option to the PiP man pages (if installed successfully).

1.4.2 `pipcc` and `pipfc`

pipcc is the compiler script for compiling PiP applications for C and C++, and **pipfc** is for Fortran, as was already mentioned in Section 1.1.1. The **--which** option will display the real back-end computer's pass. For **pipcc** or **pipfc**, users can also choose the back-end compiler by changing the environment variable **CC** or **FC**.

By default, **pipcc** and **pipfc** compile programs to create code that can execute as either a PiP task or the root process. Users can specify the **--piproot** or **--piptask** options to run an application that only supports PiP tasks or the PiP root. Any PiP software that has been compiled as PiP tasks can, in fact, run as a PiP root. The **--piptask** option is therefore the same as the **--pipboth** option (to be both root and task).

The **--cflags** option displays the actual compilation options that should be provided to the back-end compiler, and the **--lflags** option displays the linker options. The actual compiling and/or linking processes are disabled by the **--cflags** or **--lflags** options.

Listing 1.30: `pip-check` - Execution Example

```
$ pip-check /usr/bin/ps
/usr/bin/ps : not a PiP program
$ pip-check /usr/bin/ls
/usr/bin/ls : not an ELF file
$ cat /usr/bin/ls
#!/usr/bin/coreutils --coreutils-prog-shebang=ls
$ pipcc --silent pip.c -o pip
$ pip-check ./pip
./pip : Root&Task
$ pipcc --silent --piptask pip.c -o pip-task
$ pip-check ./pip-task
./pip-task : Root&Task
$ pipcc --silent --piproot pip.c -o pip-root
$ pip-check ./pip-root
./pip-root : Root
$
```

The **pip-check** application can be used to determine whether a file can be executed as a PiP program. A program may not actually run as a PiP program even if it explicitly states that it will. Linux commands cannot be run as PiP tasks, even if they are created as PIE programs. Any shell script (*shebang*, the file with the extension "#!") follows the same rules. Listing 1.30 shows an example of a shell script that uses the ls command.

1.4.3 `pip-exec`

The **pip-exec** command is used in the examples thus far to execute PiP tasks that are derivations of a single program. Though all PiP tasks created from those programs share the same address space, **pip-exec** has the ability

to invoke several programs. Programs are divided in this way by colons (:) (Listing 1.31).

Listing 1.31: `pip-exec` - Execution Example

```
$ cat prog.c
#include <pip/pip.h>
int main( int argc, char **argv ) {
  int pipid;
  pip_get_pipid( &pipid );
  printf( "This is %s [%d]\n", argv[0], pipid );
  return 0;
}
$ pipcc --silent prog.c -o a.out
$ cp a.out b.out
$ cp a.out c.out
$ pip-exec -n 2 ./a.out : -n 3 ./b.out : -n 1 ./c.out
This is ./a.out [0]
This is ./a.out [1]
This is ./b.out [2]
This is ./b.out [3]
This is ./b.out [4]
This is ./c.out [5]
$
```

1.4.4 `pips`

Similar to how the Linux ps command works, the **pips** command outputs a list of all presently active PiP tasks and PiP roots. Here is an example where three (3) **pip-exec** are launched, with each one executing one of the PiP tasks (a, b, or c).

```
$ pips
PID   TID   TT       TIME     PIP COMMAND
18741 18741 pts/0    00:00:00 RT  pip-exec
18742 18742 pts/0    00:00:00 RG  pip-exec
18743 18743 pts/0    00:00:00 RL  pip-exec
18741 18744 pts/0    00:00:00 0T  a
18745 18745 pts/0    00:00:00 0G  b
18746 18746 pts/0    00:00:00 0L  c
18747 18747 pts/0    00:00:00 1L  c
18741 18748 pts/0    00:00:00 1T  a
18749 18749 pts/0    00:00:00 1G  b
18741 18750 pts/0    00:00:00 2T  a
18751 18751 pts/0    00:00:00 2G  b
18741 18752 pts/0    00:00:00 3T  a
```

As you can see, this output resembles the on of the `ps` command quite a bit. If this is a PiP root or PiP task, it is shown in the unfamiliar `PIP`

column (first character). The other numerical digit, "0-9," stands for the PiP task, while "R" stands for root. In Section 2.2, PiP execution mode is represented by the second character.

The **pips** command has many options. Refer PiP man page (Section 1.4.1) for more details.

1.4.5 `pip-gdb`

PiP-aware `gdb` (GNU debugger) is known as **pip-gdb**. PiP tasks are introduced as inferiors of GDB. This is an illustration of a **PiP-gdb** debugging session. Note that the **pip-gdb** can only debug PiP tasks in the process mode.

```
(pip-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
```

1.4.6 `pip-mode` and `printpipmode`

The **pip-mode** command is to set PiP execution mode and the **printpip-mode** outputs the current execution mode (refer to Section 2.2 and 3.1.4).

1.4.7 `libpip.so`

The PiP library **libpip.so** can also run as a program, showing the information how the library was build and installed.

Listing 1.32: `libpip.so` - Execution Example

```
$ ${PIPLIBDIR}/libpip.so
Package:        Process-in-Processs
Version:        2.4.1
License:        the 2-clause simplified BSD License
Build OS:       Linux 5.10.104-linuxkit #1 SMP Thu Mar 17 17:08:06 UTC 2022
Build CC:       gcc (GCC) 8.5.0 20210514 (Red Hat 8.5.0-4)
Prefix dir:     /home/ahori/git/pip-2/install
PiP-glibc:      /home/ahori/pip-glibc/install/lib
ld-linux:       /home/ahori/pip-glibc/install/lib/ld-2.28.so
Commit Hash:    2485d3f923302ef03432bc52a5ddc3c4b0398fca
Debug build:    no
URL:            https://github.com/procinproc/PiP/
mailto:         procinproc-info@googlegroups.com
$
```

1.5 Summary

PiP root and PiP task

- PiP applications must be compiled using the pipfc or pipcc commands, respectively, for C, C++, and Fortran.

- The **pip-exec** command allows PiP programs to be executed as PiP tasks.
- PiP programs can be invoked as regular programs to execute as non-PiP tasks.
- Static variables in a PiP application are privatized, meaning that each PiP task has its own set of the static variables, in contrast to the traditional multi-thread paradigm (i.e. OpenMP).
- In contrast to the traditional multi-process approach (i.e. MPI), PiP tasks have access to each other's data and share the same address space.

PiP API

- The majority of PiP functions return a Linux-defined error code.
- Per address space, each PiP task has an unique **PIPID**.
- PiP root must call **pip_init()** to initialize PiP library. The initialization function may or may not be called by child PiP tasks.
- By using the **pip_spawn()** or **pip_task_spawn()** functions, the PiP root can start new PiP tasks.
- Use the **pip_named_export()** and **pip_named_import()** methods to find the address for gaining access to the data of the other PiP tasks.
- **pip_barrier_wait()** is used for tasks to synchronize. Synchronization can also be accomplished using PThreads' synchronization functions.
- The invoking PiP process and PiP root are terminated by the **pip_exit()** method.
- By using a member of the **pip_wait()** function family, the PiP root can wait for a created PiP task to finish.

1.6 Myths on PiP

I fail to distinguish what PiP performs from shared memory.

The shared memory model enables access to the data of the other process. The XPMEM[3] shared memory mode facilitates the sharing of any existing memory region with the other process, in contrast to the POSIX shared

[3] https://github.com/hpc/xpmem

memory paradigm, which only permits the sharing of newly allocated memory regions. I use the term "shared address space model" instead of "shared memory model" to describe PiP's technique for mapping memory areas. The shared address space and shared memory appear to be the same in order to access the data owned by the other.

However, the mechanisms behind the two memory theories differ greatly. To request shared memory, one must issue a system call to the OS kernel. The system calls that alter the memory mapping are rather expensive. When the addresses of the data are known, a PiP job can access any data owned by the others without making an expensive system call because PiP tasks, on the other hand, are mapped to a single memory address.

If the shared data are allocated dynamically or are distributed over the address space and difficult to pack into a memory area, the shared address space solution is advantageous.

A serious security risk might arise when many programs share the same address

A serious security risk might arise when many programs share the same address. Programs that share an address space can run thanks to PiP. Facilitating efficient information sharing between programs is the main goal here. If there is no interaction between them, running them in the PiP environment is useless.

Programs for communication fundamentally have the same outcome. Even in the most straightforward case, if two programs are connected by a Linux/Unix pipe and one of the programs crashes, the other programs also crash after receiving the `SIGPIPE` signal. Communicating processes agree on when and how to communicate with others. The PiP scenario is not exceptional.

Sharing address space makes debugging difficult

It is true that if one operation in a PiP environment destroys data that belongs to another, the consequences could be disastrous. If done maliciously, this cannot be prevented (see also above). It may be more difficult to debug the destruction if a software flaw caused it than the multi-process paradigm. There are two things to note in this situation: 1) the increased risk of actually destroying data rather than accessing an invalid memory location (`SIGSEGV`), and 2) the existence of numerous execution entities.

For the earlier point, the ASLR can be helpful. The bug's phenomena can change over time if ASLR is enabled. With the multi-thread example, the last point's situation is essentially the same.

Anyway, I haven't had any problems with issues based on this scenario yet.

My program does not have any static variables and I do not need PiP.

Programs can be written without using static variables. However, Glibc's routines use a lot of static variables. Some of the Glibc functions may be used by your runtime system. Therefore, it is generally quite difficult to write programs without using any static variables.

PiP may consume more memory than the other execution models

A single address space is used to map the memory segments of active PiP activities, as shown in Listing 3.4. For instance, Listing 3.4 only displays the memory segments connected to Glibc (`libc-2.28.so`), which is loaded three (3) times in this case to operate one PiP root and two PiP tasks, each of which depends on Glibc. A memory map of `libc-2.28.so` is contained in each segment set. The same file is used to map all three segment sets, and only one set uses the same amount of RAM.

Each address space of a process in the multi-process paradigm only has one `libc-2.28.so` segment set, although another process also uses the same memory mapping as Glibc. Therefore, practically speaking, the memory needed to run PiP tasks is comparable to the memory needed to operate multiple processes. However, regardless of the number of threads, the multi-thread model only has one variable segment that is shared by all of the threads. Additionally, performing PiP jobs requires more memory than running multiple threads does.

There must be some hidden overhead for running PiP programs

As of now, one overhead that is greater than the multi-process model is known. System calls for address space change, such `mmap()` and `brk()`. This is because any change to an address space inside the OS kernel must be locked, and lock contention causes more overhead. The scenario is the same when using the multi-thread model, and while `mmap()` has a higher overhead than the multi-process model, it is nearly identical when using the multi-thread model. Up to now, no additional overhead in PiP has been identified.

Chapter 2

PiP Advanced

So far, the basic of PiP is described, In this chapter, more detailed functionalities provides by PiP will be explained.

2.1 Rationale

The steps to spawn a PiP task are as follows (a more complete technique is provided in Section 3.1.1);

The steps to spawn a PiP task are: 1. construct a new name space by using the Linux dlmopen()() function, 2. create a PiP task process (or thread) by using the Linux clone() system call, and 3. enter the starting function of a user application.

1. build a new name space by using the Linux `dlmopen()` function,
2. create a PiP task process (or thread) by using the Linux `clone()` system call, and
3. enter the starting function of a user application.

In contrast to `dlopen()`, the `dlmopen()` function can construct a new name space. Here, the global symbol names (functions and global variables) that must be resolved upon program loading make up the name space. Functions and variables may be privatized from the other PiP tasks by establishing a distinct name space.

It's crucial to call `dlmopen()` and `clone()` in the correct order. As it looked pretty reasonable, at first I tried to call them in the following order: clone(), followed by dlmopen(), but this did not work at all. This is the rationale behind why only PiP root can start PiP tasks and watch for their terminations.

The loaded address of a program is fixed by default in some circumstances (or in the majority of circumstances prior to CentOS/Redhat 8).

If so, PiP is unable to load more than one program in the same address space. The PiP executables must be built as PIE(Position Independent Executable) files in order for the programs to be able to be loaded at any arbitrary address. All PiP tasks must be built as PIE, which means they must be built using the **--piptask** option using **pipcc** or **pipfc** (or nothing to use the default). Keep in mind that the PiP root program could not be PIE.

PiP task can be formed by running the loaded program in a new name space with another thread. Unfortunately, nothing is ever that easy. Many problems are caused by Glibc. These problems are discussed in the following section.

2.2 Execution Mode

PiP library is made to function on Linux. According to Section 2.1, it is very dependent on the `dlmopen()` and `clone()` functions. Particularly, the `clone()` function is called with a unique set of `CLONE` flags. Many Linux variants exist, and some of them do not support the specified `CLONE` flag combination (for example, McKernel[1]). There are two ways to execute PiP in such an environment: one involves executing `clone()` with a particular flag combination, and the other involves calling `pthread_create()` to start a PiP task while using the standard `CLONE` flag combination. The former is referred to as **process mode**, and the latter as **pthread mode**. Both modes keep the fundamental characteristics of the PiP, including address space sharing and variable privatization.

2.2.1 Differences Between Two Modes

`CLONE` flag combinations ultimately determine the difference in the PiP execution mode. The `CLONE` flag values `CLONE_FS`, `CLONE_FILES`, `CLONE_SIGHAND`, and `CLONE_THREAD` are reset, but `CLONE_VM` and `CLONE_SYSVSEM` are set, in contrast to the pthread mode.

Table 2.1: Differences between two modes

	Process Mode	Pthread Mode
Address Space Sharing	yes	yes
Variable Privatization	yes	yes
File Descriptors (FDs)	not shared	shared

The primary distinctions between the two modes are shown in Table 2.1. There are a lot more variations, however the PiP library offers mode-agnostic

[1] https://github.com/ihkmckernel/mckernel

functions so that users can develop PiP programs without worrying about the variations in mode.

Table 2.2: Mode-Agnostic Functions

Mode-Agnostic	Process Mode	Pthread Mode	note
pip_exit()()	`exit()`	`pthread_exit()`	termination
pip_wait()()	`wait()`	`pthread_join()`	wait termination
pip_kill()()	`kill()`	`pthread_kill()`	send signal
pip_sigmask()()	`sigprocmask()`	`pthread_sigmask()`	signal mask
pip_signal_wait()()	sigwait()	sigwait()	wait signal
pip_yield()()	`sched_yield()`	`pthread_yield()`	yield

There are also predicate functions for users to know the current mode listed in Table 2.3.

Table 2.3: Execution Mode Predicates

Function name	note
pip_is_threaded()()	if pthread mode
pip_is_shared_fd()()	if FDs are shared

In the current implementation, **pip_is_threaded()** and **pip_is_shared_fd()** have the same meaning. There may be situations in which those two terms have different meanings, so having those functions is necessary.

2.2.2 How to Specify Execution Mode

When calling **pip_init()** or setting the **PIP_MODE** environment variable at runtime, the execution mode can be selected. The function prototype for the **pip_init()** shown below. Up until this point, the first three arguments have been discussed.

```
int pip_init( int *pipidp,       [IN/OUT]
              int *ntasksp,      [IN/OUT]
              void **root_expp, [IN/OUT]
              int opts );        [IN]
```

The final opts parameter can take one of the following values: **PIP_MODE_PROCESS**, **PIP_MODE_THREAD**, or both, or zero. Zero has the same value as **PIP_MODE_PROCESS**|**PIP_MODE_PTHREAD**. The **PIP_MODE** environment can be a string that contains the words "`process`" or "`pthread`." The value of the **PIP_MODE** environment variable is checked when the opts value is zero or the value of either oring the two. If no environment is specified, the PiP library picks a suitable one. The environmental value and the ethical worth cannot be at odds with one another.

2.3 Spawning Tasks - Advanced

Further features of **pip_spawn()** and **pip_task_spawn()** that have not yet been explained will be covered in this section. The function prototypes for these functions are included below for your convenience;

```
int pip_task_spawn( pip_spawn_program_t *progp,  [IN]
                    uint32_t coreno,             [IN]
                    uint32_t opts,               [IN]
                    int *pipidp,                 [IN/OUT]
                    pip_spawn_hook_t *hookp );   [IN]

int pip_spawn( char *filename,                   [IN]
               char **argv,                      [IN]
               char **envv,                      [IN]
               int coreno,                       [IN]
               int *pipidp,                      [IN/OUT]
               pip_spawnhook_t before,           [IN]
               pip_spawnhook_t after,            [IN]
               void *hookarg );                  [IN]
```

In Section 1.2.1, the first argument of the **pip_task_spawn()** function is already covered in Section 1.2.1. To compact the first three arguments of **pip_spawn()**, use the **pip_spawn_program_t** structure.

2.3.1 Start Function

These PiP spawn routines eventually jump into the start function (`main()` or a user-specified one), as already demonstrated in Listing 1.2.1. PiP requires knowledge of the start function's address in order to enable this. Here, one of the two prerequisites must be satisfied:

- the starting function is defined as a global symbol, or
- if the starting function is defined as a local symbol then the executable file must not be stripped.

The `-rdynamic` option is used during program compilation with the **pipcc** and **pipfc** to make the symbol global for the `main()` function. When it comes to user-defined local symbols, the PiP library reads the executable file to spawn and attempts to locate the starting function using the ELF metadata. Unfortunately, if the local symbol information is stripped, it is lost, and PiP is unable to locate the starting function.

2.3.2 Stack Size

The **PIP_STACKSIZE** environment variable can be used to control the stack size of created PiP tasks. Its value can be prefixed with "`T`," "`G`,"

"M," "K," or "B" which, like the OMP_STACKSIZE environment provided by OpenMP, stands for the *TiB*, *GiB*, *MiB*, *KiB*, or *Byte* unit, respectively. *KiB* is presummated in the absence of a suffix. The environment variables KMP_STACKSIZE, GOMP_STACKSIZE, or OMP_STACKSIZE are similarly effective with the priority in this order unless **PIP_STACKSIZE** is provided. While **PIP_STACKSIZE** only impacts the stack size of PiP processes, KMP_STACKSIZE, GOMP_STACKSIZE, and OMP_STACKSIZE also affect the size of OpenMP threads.

2.3.3 CPU Core Binding

The spawned PiP task is bound to the designated CPU core by the coreno argument. This is the *N*th core number by default. The absolute core number should be ORed with the **PIP_CPUCORE_ABS** flag if users want to specify the absolute core number. Specifying this flag might not have an impact on the core number specification if the core numbers are continuous. Only certain CPU architectures with non-contiguous core numbers exhibit this disparity (e.g., Fujitsu A64FX). To connect to the same CPU cores as the root process executing the spawn function, the coreno argument might be **PIP_CPUCORE_ASIS**.

2.3.4 File Descriptors and Spawn Hooks

Similar to how fork() works, file descriptors from the root process are duplicated and provided to the newly spawned child in the **process mode**. File descriptors are simply shared by PiP root and PiP tasks in the **pthread mode**.

The file descriptor is closed after calling the **before hook** mentioned below (if any), and then the start function is called if the close-on-exec flag of a file descriptor owned by the root process is set in process mode.

The structure containing the final three arguments of **pip_spawn()** is the final argument of **pip_task_spawn()**. Calling the **pip_spawn_hook()** function will set the **pip_spawn_hook_t** structure. Here is the prototype;

```
void pip_spawn_hook( pip_spawn_hook_t *hook,   [OUT]
                     pip_spawnhook_t before,   [IN]
                     pip_spawnhook_t after,    [IN]
                     void *hookarg ) {         [IN]
typedef int (*pip_spawnhook_t)( void* );
```

Before invoking the start method (such as main()), this structure's before function is called when a PiP task is created. And when the PiP task is ready to finish, the after function is invoked. Both functions are invoked with the hookarg-specified argument, which can pass any kind of data.

In Linux/Unix, calling `fork()` and `execve()` generally results in the creation of a new process. Here, the parent process's file descriptors are transferred to the newly created child process. In many instances, such file descriptors are closed or duplicated, and other settings are changed in between calls to `fork()` and `execve()`. The task is only created by one function in PiP, hence it is not possible to use the same settings as when `fork&exec` is used. These hook functions are offered in order to accomplish this. Here is an illustration of one of these hook functions;

Listing 2.1: Before and After Hooks

```
#define _GNU_SOURCE
#include <pip/pip.h>
#include <stdlib.h>
#include <unistd.h>
#define FD_TASK          (10)
int before_hook( void *argp ) {
  int *fdp = (int*) argp;
  printf( "PID:%d Before Hook: fd=%d\n", getpid(), *fdp );
  fflush( stdout );
  dup2( 1, *fdp );
  return 0;
}
int after_hook( void *argp ) {
  int *fdp = (int*) argp;
  printf( "PID:%d After Hook:  fd=%d\n", getpid(), *fdp );
  fflush( stdout );
  close( *fdp );
  return 0;
}
int main( int argc, char **argv ) {
  int pipid, ntasks, pipid_task, i;
  int arg = FD_TASK;
  ntasks = strtol( argv[1], NULL, 10 );
  pip_init( &pipid, &ntasks, NULL, 0 );
  if( pipid == PIP_PIPID_ROOT ) {
    pip_spawn_program_t prog;
    pip_spawn_hook_t    hooks;
    pip_spawn_from_main( &prog, argv[0], argv,
                         NULL, NULL );
    pip_spawn_hook( &hooks,
                    before_hook,
                    after_hook,
                    &arg );
    printf( "PID:%d MAIN\n", getpid() );
    fflush( stdout );
    for( i=0; i<ntasks; i++ ) {
      pipid_task = i;
      pip_task_spawn( &prog, PIP_CPUCORE_ASIS,
```

```
                        0, &pipid_task, &hooks );
      pip_wait( pipid_task, NULL );
    }
  } else {
    char *msg;
    asprintf( &msg, "Hello from PIPID:%d\n", pipid );
    write( FD_TASK, msg, strlen( msg ) );
    free( msg );
  }
  pip_fin();
  return 0;
}
```

In this illustration, the **before hook** copies the root process' FD1 to FD10. Next, the job that was started uses FD10 to write a message. The **after hook** closes the FD10 finally. You should be aware that the root calls those hook functions to run them.

Listing 2.2: Before and After Hooks - Execution

```
$ ./hook 2
PID:37231 MAIN
PID:37232 Before Hook: fd=10
Hello from PIPID:0
PID:37232 After Hook:  fd=10
PID:37233 Before Hook: fd=10
Hello from PIPID:1
PID:37233 After Hook:  fd=10
$
```

2.4 Execution Context

Before I cover the following points, readers should be familiar with the execution context under PiP. The general concept of the execution context can be thought of as the CPU's state or the information in hardware registers. On PiP, this definition might not be adequate. Let's examine a case in point. Imagine that the same program, which is operating as two PiP tasks, has a function called `foo()`. By providing the function pointer and leveraging the **pip_named_export()** and **pip_named_import()**, one PiP task can call the function of another PiP task. An additional static variable, let's call `var`, is used by the function foo. When task A calls function `foo()` of task B, the called function accesses task B's variable rather than task A's (Listing 2.3 and 2.4).

Listing 2.3: Calling a Function of Another Task

```
#include <pip/pip.h>
int var;
```

```
int foo( void ) { return var; }
int main( int argc, char **argv ) {
  int pipid, ntasks, prev;
  int(*funcp)(void);
  pip_get_pipid( &pipid );
  pip_get_ntasks( &ntasks );
  prev = ( pipid == 0 ) ? ntasks - 1 : pipid - 1;
  var = pipid * 100;
  pip_named_export( foo, "foo@%d", pipid );
  pip_named_import( prev, (void**) &funcp, "foo@%d", prev );
  printf( "PIPID:%d foo(%d)=%d\n", pipid, prev, funcp() );
  return 0;
}
```

Sorry, this may detour off the main road, but in this example program, **pip_named_export()** and **pip_named_import()** are called in a different way than they were previously. Similar to Linux's `printf()`, the second and third arguments to the **pip_named_export()** and **pip_named_import()** functions, respectively, are really format strings followed by the argument(s) required by the format.

Listing 2.4: Function Call of Another Task - Execution

```
$ pip-exec -n 4 ./context
PIPID:1 foo(0)=0
PIPID:2 foo(1)=100
PIPID:3 foo(2)=200
PIPID:0 foo(3)=300
$
```

The execution context in a PiP environment may differ from that in a common sense environment, as shown in Listing 2.4, and occasionally its behavior changes in a very subtle way. This occurs pretty frequently in the PiP library and makes debugging quite challenging.

Furthermore, the CPU architecture and PIE implementation have a significant impact on the relationship between static variables and function addresses. For `x86_64` and `AArch64`, the aforementioned description is accurate, but not on `x86_32`. As a result, it is not advised to do this.

Rationale

Some readers may be curious as to why this occurs. Let me elaborate. The address map contains this ruse. Listing 2.4 displays a section of the address map with three jobs executing, with a focus on the Glibc (`/lib64/libc-2.28.so`) segments.

```
  ...
7ffff53f8000-7ffff55a4000 r-xp 00000000 fe:01 3049686    /lib64/libc-2.28.so
7ffff55a4000-7ffff57a4000 ---p 001ac000 fe:01 3049686    /lib64/libc-2.28.so
7ffff57a4000-7ffff57a8000 r--p 001ac000 fe:01 3049686    /lib64/libc-2.28.so
7ffff57a8000-7ffff57aa000 rw-p 001b0000 fe:01 3049686    /lib64/libc-2.28.so
```

```
  ...
7ffff69f6000-7ffff6ba2000 r-xp 00000000 fe:01 3049686    /lib64/libc-2.28.so
7ffff6ba2000-7ffff6da2000 ---p 001ac000 fe:01 3049686    /lib64/libc-2.28.so
7ffff6da2000-7ffff6da6000 r--p 001ac000 fe:01 3049686    /lib64/libc-2.28.so
7ffff6da6000-7ffff6da8000 rw-p 001b0000 fe:01 3049686    /lib64/libc-2.28.so
...
7ffff73d5000-7ffff7581000 r-xp 00000000 fe:01 3049686    /lib64/libc-2.28.so
7ffff7581000-7ffff7781000 ---p 001ac000 fe:01 3049686    /lib64/libc-2.28.so
7ffff7781000-7ffff7785000 r--p 001ac000 fe:01 3049686    /lib64/libc-2.28.so
7ffff7785000-7ffff7787000 rw-p 001b0000 fe:01 3049686    /lib64/libc-2.28.so
  ...
```

Glibc segments come in three sets. The final (readable and writable) segment of each set is where the static variables are located. By using the offset from the instruction (program counter relative addressing mode) to the variable, an instruction can access a static variable. In order to keep all offsets equal, the gap size between the code segment (at the top of the set) and the variable segment (at the bottom of the set) is crucial, and all gap sizes must be the same. Variables and instructions are connected in PIE[2] in this fashion, allowing PIE applications and shared objects that were created with the PIC option to be loaded in any place.

Unfortunately, not all CPU architectures implement this addressing mode. For example, `x86_32` doesn't. This architecture sacrifices one general purpose register in order to point the variable segment, causing performance to suffer as a result. Moreover, Listing 2.3 program may display a distinct behavior.

2.5 Debugging Support

While debugging PiP programs, some environment variable settings may be helpful.

2.5.1 PIP_STOP_ON_START

When this environment variable is set, the spawning PiP job is suspended (by sending `SIGSTOP` shortly before the before hook (Section 2.3.4) is called, or if before hook is not specified, the starting function is called instead. The environment's worth must conform to the following specification;

```
PIP_STOP_START=[<script-file>]@<PIPID>
```

The ⟨`PIPID`⟩ is the **PIPID** of the suspending task, and the optional ⟨`script-file`⟩ is an executable shell script to be run upon suspension. All PiP tasks that have been spawned will be stopped if "PIPID" is -1. The "script" is called with three parameters: the PID and PIPID of the suspended PiP task, then the path to the task's program. Don't forget to set the executable bit on this ⟨`script-file`⟩.

[2] This is not the case if not compiled as PIE.

Listing 2.5: Stop-on-start Script Example

```
#!/bin/sh
PID=$1
PIPID=$2
PROG=`basename $3`
echo "###" $0 "### "${PROG} ${PID} ${PIPID}
pips -f ${PID} # strace, ltrace, pip-gdb, ...
kill -CONT ${PID}
```

A sample of the PIP STOP ON START script is shown in Listing 2.5. In this case, the pips command is used in place of a debugging command[3]. It should be noted that sending the SIGSTOP signal has already stopped the target task. If you want to resume a task, you must somehow explicitly provide the SIGCONT signal to the task. Listing 2.6 displays the outcome of running this script file with the **PIP_STOP_ON_START** environment setting.

Listing 2.6: Stop-on-start Script Example - Execution

```
$ pipcc --silent hello.c -o hello
$ echo $PIP_STOP_ON_START
onstart.cmd@2
$ pip-exec -n 4 ./hello
Hello World
Hello World
PiP-INFO[36954(R):R] PiP task[2] (PID=36957) is SIGSTOPed and executing 'onstart.cmd' script
### onstart.cmd ### hello 36957 2
Hello World
  File "/home/ahori/git/pip-2/install/bin/pips", line 228
    from __future__ import print_function
    ^
SyntaxError: from __future__ imports must occur at the beginning of the file
Hello World
$
```

2.5.2 PIP_GDB_SIGNALS

When a signal provided in this environment variable is delivered to a PiP task, PiP-gdb is called. The path to PiP-gdb must be specified by the **PIP_GDB_PATH** environment. Prior to calling PiP-gdb, actions related to **PIP_SHOW_MAPS** and **PIP_SHOW_PIPS** (both will be discussed below) will be taken. This environment's format is as follows:

PIP_GDB_SIGNALS=[<SIGNAME>] { "+"|"-" <SIGNAME> }

The following is a list of possible ⟨SIGNAME⟩ values. Here, "ALL" refers to every signal listed in this table. The plus (+) and/or minus (-) symbols can be used to concatenate each signal name in this table. For instance, "ALL-SIGUSR1" denotes all signals other than SIGUSR1. SIGUSR1, SIGUSR2, and linuxdefSIGINT are represented as "SIGUSR1+SIGUSR2+SIGINT".

[3] Although ptrace() (and other commands utilizing ptrace()) were unable to run even with the cap-add=SYS_PTRACE Docker option, all examples in this paper were tested in a Docker environment (I confirmed gdb worked). In this example, **pips** was used.

Table 2.4: Possible Signal Names for **PIP_GDB_SIGNALS**

SIGHUP
SIGINT
SIGQUIT
SIGILL
SIGABRT
SIGFPE
SIGINT
SIGSEGV
SIGPIPE
SIGUSR1
SIGUSR2
ALL

2.5.3 PIP_SHOW_MAPS

The address map (Listing 3.4, for example) will be displayed if **PIP_SHOW_MAPS** environment is set to "`on`" and the signal specified by the **PIP_GDB_SIGNALS** is provided.

2.5.4 PIP_SHOW_PIPS

The **pips** command (Section 1.4.4) is invoked to display the status of the PiP tasks running in the same address space if **PIP_SHOW_PIPS** environment is set to "`on`" and a signal specified by the **PIP_GDB_SIGNALS** is delivered.

2.5.5 PIP_GDB_COMMAND

PiP-gdb will be invoked to work with this command file if the value of the **PIP_GDB_COMMAND** environment is set to a valid filename containing a series of gdb commands.

2.6 Malloc routines

Consider creating a producer-consumer program using PiP, where one PiP task serves as the producer and another as the consumer. In contrast to the traditional process approach, PiP does not require an IPC (Inter Process Communication) system call. Passing pointers pointing at data to be transmitted from the producer to the consumer is all that is necessary.

There is a problem here. If the given data was allocated by the `malloc()` function, the consumer will `free()` the passed data. As previously mentioned, each PiP task has a unique set of `malloc()` and `free()` procedures

connected to static variables that hold and manage a memory pool. When it is no longer required, the consumer tries to `free()` the data it has obtained from the producer's memory pool. Unfortunately, because the consumer's `free()` procedure is unaware of the memory space that the producer allocated, it fails (Figure 2.1). I gave this circumstance the moniker *cross-malloc-free*.

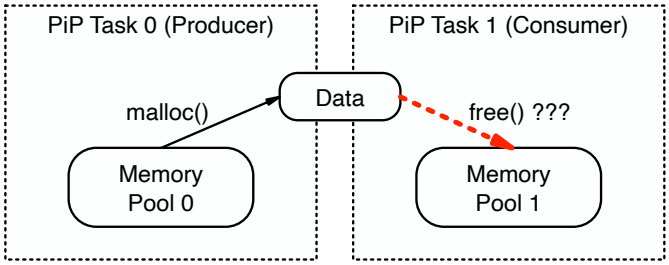

Figure 2.1: Cross-Malloc-Free Issue

I gave it a shot using the `malloc` functions that Glibc provides and discovered that while it doesn't always work, it usually does. I'm not sure why this works (again, *in most circumstances*) with the Glibc `malloc` routines, but I believe it's important to avoid this circumstance.

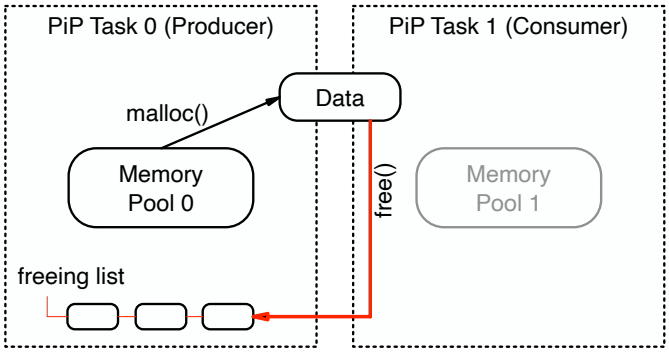

Figure 2.2: Cross-Malloc-Free with Freeing List

In order to address this, the PiP library wraps the `malloc` functions, as seen in Table 3.3. The `malloc()` wrapper function embeds the information about who allocated a memory space. When this region is to be `free()`ed, the `free()` wrapper function connects the region to the freeing list of the task allocating the region. When one of the wrapper functions for `malloc` is called, the regions in the freeing list are really `free()`ed (Figure 2.2).

2.7 XPMEM

As stated in Section 1.6, it is well known that XPMEM offers a shared memory model that is more practical than POSIX shared memory. Once more, the shared memory paradigm that XPMEM and POSIX shared memory offer is included in the shared address space that PiP offers.

The identical functions that XPMEM offers are also provided by the PiP library. Users of XPMEM-based programs can quickly convert to PiP. The XPMEM kernel module does not need to be installed while using PiP. Most notably, PiP's XPMEM operations operate far more quickly than those of XPMEM. This is due to the fact that no system call was necessary to map the memory segment of the other PiP task(s). In fact, since PiP has already had the other processes mapped from the start, the majority of XPMEM routines essentially do nothing.

Chapter 3

PiP Internals

3.1 PiP Implementation and design choices

3.1.1 Spawning Tasks

Prior to PiP version 2.4, the following steps were used to launch PiP tasks:

1. The spawning program is loaded by calling `dlmopen()`,
2. Glibc is initialized in the execution context of the loaded program,
3. Call `clone()` or `pthread_create()` (chosen by the **PIP_MODE** environment setting) to spawn the PiP task,
4. The before hook is called if any, and finally
5. Jump into the starting function.

Wrapper functions for the Glibc functions listed in Table 3.3 were first introduced in PiP version 2.4. When developing these wrapper functions, I discovered that it is nearly impossible to wrap the `dlsym()`. A function wrapper is usually implemented as; 1) obtain the wrapping function address by calling the `dlsym()` with the `RTLD_NEXT` argument, 2) do the wrapping job before and/or after calling the original function. The most of the Glibc `malloc()` routines has the other weak symbols (`malloc()` and `__libc_malloc()`, for example) and users can call the Glibc `malloc()` routines without calling `dlsym()`. If there is no such weak symbol, we cannot create a wrapper function for `dlsym()`. How can I wrap a Glibc function without calling `dlsym()`?

I used another program, dubbed **ldpip.so**, to resolve this problem. thus to load the PiP library and user application. the method for new spawning is described here;

1. Load **ldpip.so** attached in the PiP library package by calling `dlmopen()` and jump into a function defined inside of it,

2. The starting function of **ldpip.so** initializes Glibc,
3. Obtain Glibc function addresses to wrap them later by the PiP library (**libpip.so**),
4. Load **libpip.so** by calling `dlopen()`,
5. Load a user program by calling `dlopen()`,
6. Call `clone()` or `pthread_create()` (chosen by the **PIP_MODE** environment setting) to spawn the PiP task,
7. Jump into a function inside of PiP library and initialize the PiP library,
8. The before hook is called if any, and finally
9. Jump into the starting function in the user program.

No wrapper functions are defined at the moment **ldpip.so** is loaded, making it simple to access the Glibc function addresses by simply referencing them. The Glibc functions that need to be wrapped are now wrapped using the function address table built by **ldpip.so** after loading **libpip.so**.

It is necessary to run `__ctype_init()` with the execution context (Section 2.4) of the launched PiP task in order to initialize Glibc. In an earlier version of the PiP library, this was accomplished by first obtaining the initialization function by calling `dlsym()` on the loaded handle returned by `dlopen()`, and then by executing the initialization function itself. The initialization process in the new implementation consisted of calling the initialization method directly from the ldpip.so file, where the execution context was the same as that of the PiP task. So, things can go in an easier manner by introducing PiP loader program (**ldpip.so**).

3.1.2 Calling `clone()` System Call

PiP library uses a unique flag combination to call the `clone()` system call, as explained in Section 3.1.1. Since the `clone()` system call takes a lot of inputs, some of which are difficult to implement, I chose to wrap `pthread_create()` and `clone()` in order to change only the flag setting.

Wrapping the `clone()` system call is a challenge because it is used by many libraries in addition to the PiP library (e.g., PThread library). When the PiP library calls the function, the flags must be modified; however, when some other libraries call the same method, the flags should not be changed. This is a circumstance that cannot be handled by a straightforward function wrapping.

The test-and-set atomic instruction was used to construct a specific locking mechanism in order to address this problem and prevent the `clone()` system call from being invoked simultaneously from the PiP library and from

another source. The `test-and-set` instruction is used by the PiP library to lock when it is going to run `pthread_create()` with the value of current thread ID (TID), which ultimately invokes the `clone()` system call. The wrapper function of the `clone()` firstly tries to lock, but it fails with value of the current TID, then it is the case of calling from the PiP library. If the lock is successful, a different library will call it. The original `clone()` is called with the altered flags in the first scenario. In the latter scenario, the wrapper function call calls `clone()` with the same input. Naturally, the lock is released after the original `clone()` system call returns.

3.1.3 Execution Mode in Details

The PiP's **process mode** has two sub-modes: **process:preload** and **process:pipclone**[1]. Wrapping the `clone()` function mentioned above implements the **process:preload** mode. The **process:pipclone** mode implements another `pthread_create()`-like function in the **PiP-glibc** (Section 3.2.2). If the PiP library is configured to use the **PiP-glibc**, **process:pipclone** is taken; otherwise, **process:preload** is taken.

3.1.4 Name of PiP Tasks

The ability of the **pips** command (Section 1.4.4) to distinguish between PiP tasks and other customary processes and/or threads may be unclear to certain readers. Each process and thread in Linux has a name, which is visible in the COMMAND column of the top command. The `prctl()` system call (in **process mode**) or the `pthread_ setname np()` call (in **pthread mode**) are used by the PiP library to set the command name. The first two characters of the name are used by the PiP library (Table 3.1 and 3.2)).

Table 3.1: Command Name Setting (1st char.)

First char.	Distinction	Note
`R`	PiP Root	
`0..9`	PiP Task	the least significant digit of **PIPID**

PiP takes up the first two characters of the name, which can be up to 16 characters. The original command name is represented by the final 14 characters. The letters used by the pip-mode command (Section 1.4.6) to specify the PiP execution mode are shown in the abbreviation column of Table 3.2. The **pips** command may now identify the normal processes or

[1] In PiP implementation earlier than version 2.4, there was another mode **process:got**. But this becomes obsolete in the newer versions.

Table 3.2: Command Name Setting (Second char.)

2nd char.	Execution Mode	Abbreviation	Note
:	**process:preload**	L	
;	**process:pipclone**	C	
.	**process:got**	G	obsolete
|	**pthread**	T	

threads from the PiP roots and PiP tasks by using those first 2 characters of the command name.

3.2 Linux Kernel, Glibc, and Tools Problems

The problems of adopting PiP will be discussed in this section.

3.2.1 Loading a Program

Readers should have a basic understanding of how programs are loaded into memory before reading more in-depth about the Glibc difficulties when implementing PiP. Except the PiP technology, this paragraph only discusses the Linux application loading process.

When a program is launched on Linux using the `execve()` system call, the Linux kernel opens and reads the executable file while looking for the "`.interp`" ELF section.

Listing 3.1: `.interp` Section of the `ps` command

```
$ readelf -a /usr/bin/ps | grep interpreter
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
$
```

The value of the `.interp` section, /lib64/ld-linux-x86-64.so, is displayed in Listing 3.1. To load a program supplied by the `execve()` argument, the Linux kernel invokes the loader specified by the `.interp` section. The program is then loaded by the loader, along with the shared libraries needed to run it. The loader then calls the user-specified `main()`/ function after Glibc has been initialized by the starting function defined in Glibc.

Once per address space, the program loader, which is frequently referred to as `ld-linux.so`, is loaded and held in memory until the process finishes (see also Listing 3.4). By resolving external symbol references, it is in charge of any loading procedure. The Glibc functions, such as `dlopen()`, `dlmopen()`, `dlsym()`, and others, defined in the `libdl.so` (`-ldl`), are only API, and this program loader contains their functional bodies.

3.2.2 Glibc

PiP offers a novel execution model that defies classification as either a process model or a thread model. While the name of this new model has not yet been revealed, tasks in it maintain variable privatization similar to the process model while sharing the same address space as threads. The majority of the tool chains offered by Linux and others do not yet understand this execution style because it is new. In fact, the majority of the work spent developing PiP was spent trying to locate Glibc niches.

PiP Task is Unable to Spawn PiP Task

The sequence in which `dlmopen()` and `clone()` are called matters, as discussed in Section 2.1. Due to this restriction, a PiP task cannot launch another PiP task as a child task because doing so violates the constraint. As a result, calling **pip_task_spawn()** or **pip_spawn()** from a PiP task is prevented by the present PiP implementation.

Recycling PiP Tasks

According to my testing, calling the `dlclose()` function does not release the name space, loaded PIE application, or shared libraries used by the PIE program. I think fixing the Gilbc might solve this problem, but I've chosen against it. Hence, even if a PiP task ends after being established, the task's **PIPID** will not be recycled. The rationale for my choice will also be covered in Section 3.3.1.

Number of name spaces

The hard-coded limit for the number of names spaces that `dlmopen()` can create is 16. This number of 16 appears to be inadequate when taking into account the parallel execution of PiP jobs and the current CPU core count. The PiP package offers **PiP-glibc**, which offers up to 300 PiP tasks and more name spaces[2].

As the name space table is located in the `ld-linux.so`, PiP programs' `.interp` ELF section needs to be modified in order for the program to be loaded by the patched `ld-linux.so`. The GNU linker's `--dynamic-linker` option can be used to accomplish this, and the **pipcc** and **pipfc** do this.

In `ld-linux.so`, the name space table is found at the top of a structure. A few Glibc internal functions make direct references to the structure's members. This results in yet another issue. The addresses of the other members of this structure are likewise updated once the name space table's size is altered. One `ld-linux.so` can only be loaded at a time in an address

[2]Once I asked Glibc development members to increae the size, but they did not accept my opinion. Refer https://sourceware.org/bugzilla/show_bug.cgi?id=23978

space, as stated. As a result, the same Glibc must be associated with each PiP program sharing the same address.

PiP-gdb

The loaded application has information for debugging that is embedded by the `ld-linux.so` file. Unfortunately, I discovered that this code fragment is located on the pass that the kernel uses to call ld-linux.so, not on the pass that was called from `dlopen()` and `dlmopen()`. This problem was fixed with the **PiP-glibc** the patched Glibc. As a result, only PiP applications linked with **PiP-glibc** can be used with the **pip-gdb** command (Section 1.4.5).

Global lock

The majority of programs, including PiP programs, are linked using Glibc. Several PiP programs can execute simultaneously in the same address space thanks to PiP. As a result, every PiP task has a unique Glibc. Moreover, a race condition may prevent several Glibc functions from working when called simultaneously.

PiP library offers the routines **pip_glibc_lock()** and **pip_glibc_unlock()** to serialize the Gilbc function calls in order to prevent this situation. The PiP library wraps the following Glibc routines so that the lock can be introduced and users don't have to worry about the race.

Table 3.3: Glibc functions wrapped by PiP library

`dlsym`	`dlopen`	`dlmopen`
`dlinfo`	`dlclose`	`dlerror`
`dladdr`	`dlvsym`	`getaddrinfo`
`freeaddrinfo`	`gai_strerror`	`pthread_create`
`pthread_exit`		
`malloc`	`free`	`calloc`
`realloc`	`memalign`	`posix_memalign`

This table's functions `pthread_exit()` and below have function wrappers for still another reason. The malloc functions will be wrapped for the reasons that will be explained in Section 2.6, and `pthread_exit()` will be wrapped for the reasons that will be explained in Section 2.2.

These stated functions might not be all of them. Other Glibc functions may experience the race condition in some circumstances. The aforementioned locking functions can be added to eliminate this issue. Users can easily prevent deadlock situations by using this lock in a recursive fashion.

Constructors and Destructors

Programs written in C++ employ constructors and destructors. Destructor functions are typically called as the program is about to end, whereas constructor functions are typically called right before the program starts.

The behavior of the constructors and destroyers differs slightly in PiP. I should begin by describing the generic implementation of constructors and destructors in order to clarify this. The `.init_array` section of an ELF file contains a list of the constructor functions. The `.fini_array` section contains a list of the destructor functions. When `ld-linux.so` has finished loading and linking objects, constructors are called. When `_exit()` or `dlclose()` is called, destructors are invoked.

Back to PiP now. When starting a PiP process, constructors are once more called inside of the `dlmopen()` call. The PiP root process calls the `dlmopen()` function. Hence, the root of a program calls the constructors. Here is the example;

Listing 3.2: Constructors and Destructors

```
#include <pip/pip.h>
#include <iostream>
#include <unistd.h>
#include <sys/types.h>
char *pipidstr( void ) {
  static char idstr[32];
  int pipid;
  if( pip_get_pipid( &pipid ) != 0 ) {
    sprintf( idstr, "[R] PID:" );
  } else {
    sprintf( idstr, "[%d] PID:", pipid );
  }
  return idstr;
}
static int x = 0;
class Hello {
public:
  Hello(void ) {
    std::cout << pipidstr() << getpid() << " Hello" <<
      std::endl;
  }
  ~Hello(void ) {
    std::cout << pipidstr() << getpid() << " Bye" <<
      std::endl;
  }
};
Hello hello;
int main() {
  std::cout << pipidstr() << getpid() << " MAIN " <<
    std::endl;
```

```
  return 0;
}
```

A C++ program with a constructor and destructor is shown in Listing 3.2. The PiP library has not yet been initialized when the constructor of this program is invoked, and **pip_get_pipid()** returns an error (`EPERM`). Hence, the `pipidstr()` function handles this circumstance. The executable example of this program is shown in Listing 3.3. As can be seen, the PIDs produced by the constructors differ from those of the PiP tasks.

Listing 3.3: Constructors and Destructors - Execution

```
$ ./hello
[R] PID:37019 Hello
[R] PID:37019 MAIN
[R] PID:37019 Bye
$ pip-exec -n 2 ./hello
[R] PID:37020 Hello
[0] PID:37021 MAIN
[0] PID:37021 Bye
[R] PID:37020 Hello
[1] PID:37022 MAIN
[1] PID:37022 Bye
$
```

`LD_PRELOAD`

PiP tasks cannot be used with `LD_PRELOAD`; only PiP root may. This is due to the fact that dlmopen() flatly disregards the `LD_PRELOAD` environment variable.

Shared Objects

Certain shared objects, including runtime libraries related to GCC, need to be in the same directory as the `ld-linux.so`. To adhere to the restriction, the **PiP-glibc** package's **piplnlibs** shell script creates symbolic links of the shared objects in the `/lib64` directory.

Loading Program by `dlmopen`

A PIE program cannot be loaded using the `dlmopen()` function in CentOS/RedHat 8 (or maybe newer ones)[3]. This Glibc restriction is to be circumvented via the **pip-unpie** tool. Instead of being called directly by users, this program is run automatically by the **pipcc** or **pipfc** when a PiP executable is created.

[3]Refer https://sourceware.org/bugzilla/show_bug.cgi?id=11754#c15

3.2.3 Glibc RPATH Setting

Every program, including PiP, has its RPATHs added automatically when using Spack[4]. While using Spack to install PiP-glibc, an issue occurs. Spack adds the RPATH value to the compiled **PiP-glibc** but loading Glibc with the RPATH setting is not allowed in CentOS/Redhat 8. **PiP-glibc** has a program (**annul_rpath**) to unset the RPATH setting of the **PiP-glibc** that has been built in order to prevent this.

3.2.4 Linux

Heap Segment

There is another issue which comes from Linux kernel, not from Glibc. Before explaining this issue, let us start from the how an address space is composed.

Listing 3.4: A Memory Map Example

```
00400000-00402000 r-xp 00000000 00:71 77130812           /PiP/bin/pip-exec
00601000-00602000 r--p 00001000 00:71 77130812           /PiP/bin/pip-exec
00602000-00603000 rw-p 00002000 00:71 77130812           /PiP/bin/pip-exec
00603000-00624000 rw-p 00000000 00:00 0  [heap]
7fffe8000000-7fffe8021000 rw-p 00000000 00:00 0
7fffe8021000-7fffec000000 ---p 00000000 00:00 0
7fffeefff000-7fffef000000 ---p 00000000 00:00 0
7fffef000000-7ffff0000000 rwxp 00000000 00:00 0
7ffff0000000-7ffff0021000 rw-p 00000000 00:00 0
7ffff0021000-7ffff4000000 ---p 00000000 00:00 0
7ffff46da000-7ffff46db000 r-xp 00000000 00:71 79448679   /PiP/example/a.out
7ffff46db000-7ffff48da000 ---p 00001000 00:71 79448679   /PiP/example/a.out
7ffff48da000-7ffff48db000 r--p 00000000 00:71 79448679   /PiP/example/a.out
7ffff48db000-7ffff48dc000 rw-p 00001000 00:71 79448679   /PiP/example/a.out
7ffff48dc000-7ffff48f6000 r-xp 00000000 00:71 77130790   /PiP/lib/libpip.so.0
7ffff48f6000-7ffff4af6000 ---p 0001a000 00:71 77130790   /PiP/lib/libpip.so.0
7ffff4af6000-7ffff4af7000 r--p 0001a000 00:71 77130790   /PiP/lib/libpip.so.0
7ffff4af7000-7ffff4af8000 rw-p 0001b000 00:71 77130790   /PiP/lib/libpip.so.0
7ffff4af8000-7ffff4bf8000 rw-p 00000000 00:00 0
7ffff4bf8000-7ffff4da4000 r-xp 00000000 fe:01 3445390    /lib64/libc-2.28.so
7ffff4da4000-7ffff4fa4000 ---p 001ac000 fe:01 3445390    /lib64/libc-2.28.so
7ffff4fa4000-7ffff4fa8000 r--p 001ac000 fe:01 3445390    /lib64/libc-2.28.so
7ffff4fa8000-7ffff4faa000 rw-p 001b0000 fe:01 3445390    /lib64/libc-2.28.so
7ffff4faa000-7ffff4fae000 rw-p 00000000 00:00 0
7ffff4fae000-7ffff4fc5000 r-xp 00000000 fe:01 3446072    /lib64/libpthread-2.28.so
7ffff4fc5000-7ffff51c4000 ---p 00017000 fe:01 3446072    /lib64/libpthread-2.28.so
7ffff51c4000-7ffff51c5000 r--p 00016000 fe:01 3446072    /lib64/libpthread-2.28.so
7ffff51c5000-7ffff51c6000 rw-p 00017000 fe:01 3446072    /lib64/libpthread-2.28.so
7ffff51c6000-7ffff51ca000 rw-p 00000000 00:00 0
7ffff51ca000-7ffff51cc000 r-xp 00000000 fe:01 3445775    /lib64/libdl-2.28.so
7ffff51cc000-7ffff53cc000 ---p 00002000 fe:01 3445775    /lib64/libdl-2.28.so
7ffff53cc000-7ffff53cd000 r--p 00002000 fe:01 3445775    /lib64/libdl-2.28.so
7ffff53cd000-7ffff53ce000 rw-p 00003000 fe:01 3445775    /lib64/libdl-2.28.so
7ffff53ce000-7ffff53d6000 r-xp 00000000 00:71 77130791   /PiP/lib/ldpip.so.0
7ffff53d6000-7ffff55d5000 ---p 00008000 00:71 77130791   /PiP/lib/ldpip.so.0
7ffff55d5000-7ffff55d6000 r--p 00007000 00:71 77130791   /PiP/lib/ldpip.so.0
7ffff55d6000-7ffff55d7000 rw-p 00008000 00:71 77130791   /PiP/lib/ldpip.so.0
7ffff55d7000-7ffff55d8000 ---p 00000000 00:00 0
7ffff55d8000-7ffff65d8000 rwxp 00000000 00:00 0
7ffff65d8000-7ffff65d9000 r-xp 00000000 00:71 79448679   /PiP/example/a.out
7ffff65d9000-7ffff67d8000 ---p 00001000 00:71 79448679   /PiP/example/a.out
7ffff67d8000-7ffff67d9000 r--p 00000000 00:71 79448679   /PiP/example/a.out
7ffff67d9000-7ffff67da000 rw-p 00001000 00:71 79448679   /PiP/example/a.out
7ffff67da000-7ffff67f4000 r-xp 00000000 00:71 77130790   /PiP/lib/libpip.so.0
7ffff67f4000-7ffff69f4000 ---p 0001a000 00:71 77130790   /PiP/lib/libpip.so.0
7ffff69f4000-7ffff69f5000 r--p 0001a000 00:71 77130790   /PiP/lib/libpip.so.0
7ffff69f5000-7ffff69f6000 rw-p 0001b000 00:71 77130790   /PiP/lib/libpip.so.0
7ffff69f6000-7ffff6ba2000 r-xp 00000000 fe:01 3445390    /lib64/libc-2.28.so
7ffff6ba2000-7ffff6da2000 ---p 001ac000 fe:01 3445390    /lib64/libc-2.28.so
7ffff6da2000-7ffff6da6000 r--p 001ac000 fe:01 3445390    /lib64/libc-2.28.so
```

[4] https://spack.io

```
7ffff6da6000-7ffff6da8000 rw-p 001b0000 fe:01 3445390    /lib64/libc-2.28.so
7ffff6da8000-7ffff6dac000 rw-p 00000000 00:00 0
7ffff6dac000-7ffff6dc3000 r-xp 00000000 fe:01 3446072    /lib64/libpthread-2.28.so
7ffff6dc3000-7ffff6fc2000 ---p 00017000 fe:01 3446072    /lib64/libpthread-2.28.so
7ffff6fc2000-7ffff6fc3000 r--p 00016000 fe:01 3446072    /lib64/libpthread-2.28.so
7ffff6fc3000-7ffff6fc4000 rw-p 00017000 fe:01 3446072    /lib64/libpthread-2.28.so
7ffff6fc4000-7ffff6fc8000 rw-p 00000000 00:00 0
7ffff6fc8000-7ffff6fca000 r-xp 00000000 fe:01 3445775    /lib64/libdl-2.28.so
7ffff6fca000-7ffff71ca000 ---p 00002000 fe:01 3445775    /lib64/libdl-2.28.so
7ffff71ca000-7ffff71cb000 r--p 00002000 fe:01 3445775    /lib64/libdl-2.28.so
7ffff71cb000-7ffff71cc000 rw-p 00003000 fe:01 3445775    /lib64/libdl-2.28.so
7ffff71cc000-7ffff71d4000 r-xp 00000000 00:71 77130791   /PiP/lib/ldpip.so.0
7ffff71d4000-7ffff73d3000 ---p 00008000 00:71 77130791   /PiP/lib/ldpip.so.0
7ffff73d3000-7ffff73d4000 r--p 00007000 00:71 77130791   /PiP/lib/ldpip.so.0
7ffff73d4000-7ffff73d5000 rw-p 00008000 00:71 77130791   /PiP/lib/ldpip.so.0
7ffff73d5000-7ffff7581000 r-xp 00000000 fe:01 3445390    /lib64/libc-2.28.so
7ffff7581000-7ffff7781000 ---p 001ac000 fe:01 3445390    /lib64/libc-2.28.so
7ffff7781000-7ffff7785000 r--p 001ac000 fe:01 3445390    /lib64/libc-2.28.so
7ffff7785000-7ffff7787000 rw-p 001b0000 fe:01 3445390    /lib64/libc-2.28.so
7ffff7787000-7ffff778b000 rw-p 00000000 00:00 0
7ffff778b000-7ffff77a2000 r-xp 00000000 fe:01 3446072    /lib64/libpthread-2.28.so
7ffff77a2000-7ffff79a1000 ---p 00017000 fe:01 3446072    /lib64/libpthread-2.28.so
7ffff79a1000-7ffff79a2000 r--p 00016000 fe:01 3446072    /lib64/libpthread-2.28.so
7ffff79a2000-7ffff79a3000 rw-p 00017000 fe:01 3446072    /lib64/libpthread-2.28.so
7ffff79a3000-7ffff79a7000 rw-p 00000000 00:00 0
7ffff79a7000-7ffff79a9000 r-xp 00000000 fe:01 3445775    /lib64/libdl-2.28.so
7ffff79a9000-7ffff7ba9000 ---p 00002000 fe:01 3445775    /lib64/libdl-2.28.so
7ffff7ba9000-7ffff7baa000 r--p 00002000 fe:01 3445775    /lib64/libdl-2.28.so
7ffff7baa000-7ffff7bab000 rw-p 00003000 fe:01 3445775    /lib64/libdl-2.28.so
7ffff7bab000-7ffff7bc5000 r-xp 00000000 00:71 77130790   /PiP/lib/libpip.so.0
7ffff7bc5000-7ffff7dc5000 ---p 0001a000 00:71 77130790   /PiP/lib/libpip.so.0
7ffff7dc5000-7ffff7dc6000 r--p 0001a000 00:71 77130790   /PiP/lib/libpip.so.0
7ffff7dc6000-7ffff7dc7000 rw-p 0001b000 00:71 77130790   /PiP/lib/libpip.so.0
7ffff7dc7000-7ffff7dea000 r-xp 00000000 fe:01 3446237    /lib64/ld-2.28.so
7ffff7ed0000-7ffff7fe4000 rw-p 00000000 00:00 0
7ffff7fe4000-7ffff7fe8000 r--p 00000000 00:00 0          [vvar]
7ffff7fe8000-7ffff7fea000 r-xp 00000000 00:00 0          [vdso]
7ffff7fea000-7ffff7feb000 r--p 00023000 fe:01 3446237    /lib64/ld-2.28.so
7ffff7feb000-7ffff7fff000 rw-p 00024000 fe:01 3446237    /lib64/ld-2.28.so
7ffffffde000-7ffffffff000 rwxp 00000000 00:00 0          [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0  [vsyscall]
```

Listing 3.4 shows an example of the output of doing "`cat /proc/⟨PID⟩/maps`." Here, "`pip-exec -n 2 ./a.out`" was executed, resulting one **pip-exec** process and two `./a.out` tasks. The file, `/proc/⟨PID⟩/maps`, lists all memory segments in an address space of the process ⟨`PID`⟩ usually. A loaded shared object has consecutive three or four segments; executable, gap (not accessible, if any), constants and data. The rightmost column of a line indicates the `mmap()`ed filename, the second from the left column indicates the permission of the memory segment. '`r`' is readable, '`w`' is writable, '`x`' is executable and '`p`' is private (copy-on-write). There are also some special segments whose filename is in a pair of square brackets; `[stack]`, `[heap]`, and so on. These are created by the Linux kernel for special purposes as their names suggest. The segments having no filename are created by the `mmap()` system call.

Remember, this is the address space of running one PiP root and two PiP tasks, resulting to have all the segments of the three tasks. Note that the all three tasks have exactly the same `/proc/⟨PID⟩/maps` content, and there are three sets of a shared library and only one set of `ld-linux.so` (`ld-2.28.so`) can be seen in Listing 3.4.

Usually, the heap segment, mainly used by `malloc()`, exists only one per address space. As shown in this example, there is only one heap segment, meaning the heap segment is shared by three tasks (one for root and two for PiP tasks).

The size of the heap segment can be increased or decreased by calling the `brk()` system call. Most cases, there are two calls of `brk()` to allocate or deallocate heap memory, one for obtaining the current heap end address and another for setting the new heap end address. This is exactly what the Glibc's `sbrk()` does. Apparently, this API is not thread-safe at all, and thus, the shared heap memory cannot be used safely by PiP tasks.

Fortunately, the `malloc()` routines in Glibc is designed to check if there are two or more name spaces and if so they do not use the `brk()` system call, use `mmap()` instead. So, the Glibc `malloc()` routines can work with PiP without any problem. However, if some other routines use the `brk()` system call (or `sbrk()` Glibc function), for example, replacing the Glibc `malloc()` routines with some other `malloc()` implementation, then this shared heap may result in a problem.

Core File

Suppose that we have a catastrophic situation and all PiP tasks and their root process dump core files of their own. On the current Linux, a core file is associated with a process (including threads inside of it). Thus, each PiP task and the root may produce core, resulting to have many core files. Here, the address space of them are shared and the created core files and all of them are almost the same excepting the CPU state.

Let me explain this with an example. Suppose that we have PiP task A and B running on the same address space of the root P, and an error happens resulting all P, A, and B produce core files. There can be a small time difference when to produce each core file. When the first core file, of A for instance, is being created, the other P and B are still running and the memory of the shared address space can be altered by those running tasks. `gdb`, however, assumes that a core file is a consistent snapshot of memory and CPU state. The above PiP situation breaks this assumption. If B produces another core file, may or may not be caused by the error on A, the same situation can happen. Thus, the **pip-gdb** command (Section 1.4.5) does not support for debugging from a core file. To solve this issue, PiP-aware OS kernel to have the consistent core files is needed.

3.2.5 Tools

As described, PiP sets a special combination of the `clone()` flags. As a result of this, some tools do not work. Here is the list of tools which are known to work or not at the time of this writing[5].

[5] `ltrace` depends on its version

Table 3.4: Compatibility of Tools

Compatible	Incompatible
`strace`	`valgrind`
`ltrace`	

3.3 Remaining Issues

3.3.1 Retrieving Memory

Let us suppose a case where PiP task A pass a pointer to PiP task B (Listing 1.8, for example). After then, task A terminates for some reason. What if task B tries to dereference the pointer to access data which task A had? This situation can also happen if the string obtained by calling `getenv()` is passed to the other task. The consequence of this may introduce difficult situation hard to debug. This situation must be detected by compilers and/or tools which are aware of PiP-style execution model.

So, I decided not to reclaim any memory resources when a task terminates, not calling `dlclose()` nor `free()`. In the current PiP implementation, **PIPID** can be allocated only once. And not releasing memory resource will not cause further problem.

Chapter 4

PiP Installation

There are several ways to install PiP listed below;

- Building from source code
- **pip-pip** command
- Using *Spack*

It was scheduled to use RPM (yum) and Docker, but they are not available at the time of this writing.

4.1 Building from Source Code

Usually, building full PiP package consists of the following steps;

1. Building **PiP-glibc** (optional)
2. Building PiP library
3. Building **PiP-gdb** (optional)

The 1 and 3 steps are optional, but **PiP-gdb** requires **PiP-glibc**. So the possible combinations are;

- PiP library only
- **PiP-glibc**and PiP library, and
- **PiP-glibc**, PiP library and **PiP-gdb**.

Listing 4.1 shows a typical case of building full set of PiP software package.

Listing 4.1: Building from Source Code

```
$ PIPTOP=$PWD
$ git clone https://github.com/procinproc/PiP-glibc.git
  ...
$ mkdir glibc-build
$ pushd glibc-build
$ ../PiP-glibc/build.sh ${PIPTOP}/install
  ...
$ popd
$ git clone https://github.com/procinproc/PiP.git
  ...
$ pushd PiP
$ ./configure --prefix=${PIPTOP}/install --with-glibc-libdir=${
    PIPTOP}/install/lib
  ...
$ make install
  ...
$ popd
$ git clone https://github.com/procinproc/PiP-Testsuite.git
  ...
$ pushd PiP-Testsuite
$ ./configure --with-pip=${PIPTOP}/install
  ...
$ make test
  ...
$ popd
$ git clone https://github.com/procinproc/PiP-gdb.git
  ...
$ pushd PiP-gdb
$ ./build.sh --prefix=${PIPTOP}/install --with-pip=${PIPTOP}/
    install
  ...
$ ./test.sh
  ...
$ popd
$
```

4.2 pip-pip command

The procedure to install full set of PiP package might be cumbersome, but installing PiP package by using the **pip-pip** (https://github.com/procinproc/PiP-pip) command is much easier.

Listing 4.2: PiP-pip installation example

```
$ git clone https://github.com/procinproc/PiP-pip.git
$ cd PiP-pip
$ ./pip-pip --yes

RedHat/CentOS:           8
CPU Architecture:        x86_64
```

```
List of installations
GITHUB  PiP-v2
  Prefix dir:     ${PWD}/install/x86_64_centos-8_github_pip-2
  Work dir:       ${PWD}/work/x86_64_centos-8_github_pip-2

  .....

Summary
OK      git https://github.com/procinproc/PiP.git@pip-2 ${PWD}/
    install/x86_64_centos-8_github_pip-2
$
```

4.3 Using Spack

Spack[1] is another installation tool designed for the HPC software packages and PiP can also be installed by using Spack. Listing 4.3 shows the example of installing PiP (including PiP-glibc)[2].

Listing 4.3: Spack installation example

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

[1] https://spack.io

[2] Unfortunately, the current version does not install PiP-gdb for some reason.

Bibliography

[1-Hori18] Atsushi Hori, Min Si, Balazs Gerofi, Masamichi Takagi, Jai 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, page 131143, New York, NY, USA, 2018. Association for Computing Machinery. (**Note: This is the first paper proposing Process-in-Process.**).

URL `https://doi.org/10.1145/3208040.3208045`

Abstract: The two most common parallel execution models for many-core CPUs today are multiprocess (e.g., MPI) and multithread (e.g., OpenMP). The multiprocess model allows each process to own a private address space, although processes can explicitly allocate shared-memory regions. The multithreaded model shares all address space by default, although threads can explicitly move data to thread-private storage. In this paper, we present a third model called process-in-process (PiP), where multiple processes are mapped into a single virtual address space. Thus, each process still owns its process-private storage (like the multiprocess model) but can directly access the private storage of other processes in the same virtual address space (like the multithread model). The idea of address-space sharing between multiple processes itself is not new. What makes PiP unique, however, is that its design is completely in user space, making it a portable and practical approach for large supercomputing systems where porting existing OS-based techniques might be hard. The PiP library is compact and is designed for integrating with other runtime systems such as MPI and OpenMP as a portable low-level support for boosting communication performance in HPC applications.

We showcase the uniqueness of the PiP environment through both a variety of parallel runtime optimizations and direct use in a data analysis application. We evaluate PiP on several platforms including two high-ranking supercomputers, and we measure and analyze the performance of PiP by using a variety of micro- and macro-kernels, a proxy application as well as a data analysis application.

[2-Hori20] A. Hori, B. Gerofi, and Y. Ishikawa. An implementation of user-level processes using address space sharing. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 976–984, 2020. (**Note: This is the second paper on PiP experimental version (v3) implementing Bi-Level Thread and User-Level Process.**).

Abstract: There is a wide range of implementation approaches to multi-threading. User-level threads are efficient because threads can be scheduled by a user-defined scheduling policy that suits the needs of the specific application. However, user-level threads are unable to handle blocking system-calls efficiently. To the contrary, kernel-level threads incur large overhead during context switching. Kernel-level threads are scheduled by the scheduling policy provided by the OS kernel which is hard to customize to application needs. We propose a novel thread execution model, *bi-level thread*, that combines the best aspects of the two conventional thread implementations. A bi-level thread can be either a kernel-level thread or a user-level thread at runtime. Consequently, the context switching overhead of a bi-level thread is as low as that of user-level threads, but thread scheduling can be defined by user policies. Blocking system-calls, on the other hand, can be called as a kernel-level thread without blocking the execution of other user-level threads. Furthermore, the proposed bi-level thread is combined with an address space sharing technique which allows processes to share the same virtual address space. Processes sharing the same address space can be scheduled with the same technique as user-level threads, thus we call this implementation a *user-level process*. However, the main difference between threads and processes is that threads share most of the kernel state of the underlying process, such as

process ID and file descriptors, whereas different processes do not. A user-level process must guarantee that the system-calls always access the appropriate kernel information that belongs to the particular process. We call this *system-call consistency*. In this paper, we show that the proposed bi-level threads, implemented in an address space sharing library, can resolve the blocking system-call issue of user-level threads, while at the same time it retains system-call consistency for the user-level process. A prototype implementation, ULP-PiP, proves these concepts and the basic performance of the prototype is evaluated. Evaluation results using asynchronous I/O indicate that the overlap ratio of our implementation outperforms that in Linux.

[3-Ouyang20] Kaiming Ouyang, Min Si, Atsushi Hori, Zizhong Chen, and Pavan Balaji. Cab-mpi: Exploring interprocess work-stealing towards balanced mpi communication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020. (**Note: This is the first MPI optimization paper using PiP.**).

Abstract: Load balance is essential for high-performance applications. Unbalanced communication can cause severe performance degradation, even in computation-balanced BSP applications. Designing communication-balanced applications is challenging, however, because of the diverse communication implementations at the underlying runtime system. In this paper, we address this challenge through an interprocess work-stealing scheme based on process-memory-sharing techniques. We present CAB-MPI, an MPI implementation that can identify idle processes inside MPI and use these idle resources to dynamically balance communication workload on the node. We design throughput-optimized strategies to ensure efficient stealing of the data movement tasks. We demonstrate the benefit of work stealing through several internal processes in MPI, including intranode data transfer, pack/unpack for noncontiguous communication, and computation in one-sided accumulates. The implementation is evaluated through a set of microbenchmarks and proxy applications on Intel Xeon and Xeon Phi platforms.

[4-Ouyang21] Kaiming Ouyang, Min Si, Astushi Hori, Zizhong Chen, and Pavan Balaji. Daps: A dynamic asynchronous progress stealing model for mpi communication. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 516–527, 2021. (**Note: This is the another MPI optimization paper using PiP.**).

Abstract: MPI provides nonblocking point-to-point and one- sided communication models to help applications achieve com- munication and computation overlap. These models provide the opportunity for MPI to offload data transfer to low level network hardware while the user process is computing. In practice, however, MPI implementations have to often handle complex data transfer in software due to limited capability of network hardware. Therefore, additional asynchronous progress is necessary to ensure prompt progress of these software- handled communication. Traditional mechanisms either spawn an additional background thread on each MPI process or launch a fixed number of helper processes on each node. Both mechanisms may degrade performance in user computation due to statically occupied CPU resources. The user has to fine-tune the progress resource deployment to gain overall performance. For complex multiphase applications, unfortunately, severe per- formance degradation may occur due to dynamically changing communication characteristics and thus changed progress re- quirement. This paper proposes a novel Dynamic Asynchronous Progress Stealing model, called Daps, to completely address the asynchronous progress complication. Daps is implemented inside the MPI runtime. It dynamically leverages idle MPI processes to steal communication progress tasks from other busy computing processes located on the same node. The basic concept of Daps is straightforward; however, various implementation challenges have to be resolved due to the unique requirements of interprocess data and code sharing. We present our design that ensures high performance while maintaining strict program correctness. We compare Daps with state-of-the-art asynchronous progress approaches by utilizing both microbenchmarks and HPC proxy applications.

[5-Hori22] Atsushi Hori, Kaiming Ouyang, Balazs Gerofi, and Yutaka Ishikawa. On thedifference between shared memory andshared address space inhpc communication. In Dhabaleswar K. Panda and Michael Sullivan, editors, *Supercomputing Frontiers*, pages 59–78, Cham, 2022. Springer International Publishing. (**Note: This is the third paper on PiP comparing shared memory model and shared address space model which PiP provides.**).

Abstract: Shared memory mechanisms, e.g., POSIX shmem or XPMEM, are widely used to implement efficient intra-node communication among processes running on the same node. While POSIX shmem allows other processes to access only newly allocated memory, XPMEM allows accessing any existing data and thus enables more efficient communication because the send buffer content can directly be copied to the receive buffer. Recently, the shared address space model has been proposed, where processes on the same node are mapped into the same address space at the time of process creation, allowing processes to access any data in the shared address space. Process-in-Process (PiP) is an implementation of such mechanism. The functionalities of shared memory mechanisms and the shared address space model look very similar – both allow accessing the data of other processes –, however, the shared address space model includes the shared memory model. Their internal mechanisms are also notably different. This paper clarifies the differences between the shared memory and the shared address space models, both qualitatively and quantitatively. This paper is not to showcase applications of the shared address space model, but through minimal modifications to an existing MPI implementation it highlights the basic differences between the two models. The following four MPI configurations are evaluated and compared; 1) POSIX Shmem, 2) XPMEM, 3) PiP-Shmem, where intra-node communication is implemented to utilize POSIX shmem but MPI processes share the same address space, and 4) PiP-XPMEM, where XPMEM functions are implemented by the PiP library (without the need for linking to XPMEM library). Evaluation is done using the Intel MPI benchmark suite and six HPC benchmarks (HPCCG,

miniGhost, LULESH2.0, miniMD, miniAMR and mpiGraph). Most notably, mpiGraph performance of PiP-XPMEM outperforms the XPMEM implementation by almost 1.5x. The performance numbers of HPCCG, miniGhost, miniMD, LULESH2.0 running with PiP-Shmem and PiP-XPMEM are comparable with those of POSIX Shmem and XPMEM. PiP is not only a practical implementation of the shared address space model, but it also provides opportunities for developing new optimization techniques, which the paper further elaborates on.

[6-Ouyang22] Kaiming Ouyang. PhD thesis, University of California Riverside, 2022. (**Dr. Ouyang's Ph.D. Thesis using PiP.**).

Abstract: In exascale computing era, applications are executed at larger scale than ever before, which results in higher requirement of scalability for communication library design. Message Pass- ing Interface (MPI) is widely adopted by the parallel application nowadays for interprocess communication, and the performance of the communication can significantly impact the overall performance of applications especially at large scale. There are many aspects of MPI communication that need to be explored for the maximal message rate and network throughput. Considering load balance, communication load balance is essential for high-performance applications. Unbalanced communication can cause severe performance degradation, even in computation-balanced Bulk Synchronous Parallel (BSP) applications. MPI communi- cation imbalance issue is not well investigated like computation load balance. Since the communication is not fully controlled by application developers, designing communication-balanced applications is challenging because of the diverse communication implementations at the underlying runtime system. In addition, MPI provides non-blocking point-to-point and one-sided communica- tion models where asynchronous progress is required to guarantee the completion of MPI communications and achieve better communication and computation overlap. Traditional mechanisms either spawn an additional background thread on each MPI process or launch a fixed number of helper processes on each node. For

complex multiphase applications, unfortunately, severe performance degradation may occur due to dynamically changing communication characteristics. On the other hand, as the number of CPU cores and nodes adopted by the ap- plications greatly increases, even the small message size MPI collectives can result in the huge communication overhead at large scale if they are not carefully designed. There are MPI collective algorithms that have been hierarchically designed to saturate inter-node network bandwidth for the maximal communication performance. Meanwhile, advanced shared memory techniques such as XPMEM, KNEM and CMA are adopted to accelerate intra-node MPI collective communication. Unfortunately, these studies mainly focus on large-message collective optimization which leaves small- and medium-message MPI collec- tives suboptimal. In addition, they are not able to achieve the optimal performance due to the limitations of the shared memory techniques. To solve these issues, we first present CAB-MPI, an MPI implementation that can identify idle processes inside MPI and use these idle resources to dynamically balance com- munication workload on the node. We design throughput-optimized strategies to ensure efficient stealing of the data movement tasks. The experimental results show the benefits of CAB-MPI through several internal processes in MPI, including intranode data transfer, pack/unpack for noncontiguous communication, and computation in one-sided accumulates through a set of microbenchmarks and proxy applications on Intel Xeon and Xeon Phi plat- forms. Then, we propose a novel Dynamic Asynchronous Progress Stealing model (Daps) to completely address the asynchronous progress complication; Daps is implemented inside the MPI runtime, and it dynamically leverages idle MPI processes to steal communication progress tasks from other busy computing processes located on the same node. We compare Daps with state-of-the-art asynchronous progress approaches by utilizing both microbench- marks and HPC proxy applications, and the results show the Daps can outperform the baselines and achieve less idleness during asynchronous communication. Finally, to fur- ther improve MPI collectives performance, we propose Process-in-Process

based Multiobject Interprocess MPI Collective (PiP-MColl) design to maximize small and medium-message MPI collective performance at a large scale. Different from previous studies, PiP-MColl is designed with efficient multiple senders and receivers collective algorithms and adopts Process-in-Process shared memory technique to avoid unnecessary system call and page fault overhead to achieve the best intra- and inter-node message rate and throughput. We focus on three widely used MPI collectives MPI Scatter, MPI Allgather and MPI Allreduce and apply PiP-MColl to them. Our microbenchmark and real-world HPC application ex- perimental results show PiP-MColl can significantly improve the collective performance at a large scale compared with baseline PiP-MPICH and other widely used MPI libraries such as OpenMPI, MVAPICH2 and Intel MPI.

Index

ASLR, 25

close-on-exec, 31

Linux Command
  gdb, 36, 50
  ltrace, 50
Linux Define Symbol
  CLONE_FILES, 28
  CLONE_FS, 28
  CLONE_SIGHAND, 28
  CLONE_SYSVSEM, 28
  CLONE_THREAD, 28
  CLONE_VM, 28
  EPERM, 9
  RTLD_NEXT, 40
  SIGCONT, 36
  SIGPIPE, 25
  SIGSEGV, 25
  SIGSTOP, 35, 36
  SIGUSR1, 36
  SIGUSR2, 36
  WEXITSTATUS, 13
  WIFEXITED, 13
  WIFSIGNALED, 13
  WTERMISIG, 13
Linux Environment Variable
  GOMP_STACKSIZE, 31
  KMP_STACKSIZE, 31
  LD_PRELOAD, 47
  OMP_STACKSIZE, 31
Linux Function
  __ctype_init, 41
  __libc_malloc, 40
  _exit, 46
  brk, 26, 50
  clone, 27, 28, 40–42, 44, 50
  dlclose, 44, 46
  dlmopen, 27, 28, 40, 43–47
  dlopen, 27, 41, 43, 45
  dlsym, 40, 41, 43
  execve, 8, 32, 43
  exit, 15
  fork, 31, 32
  free, 37, 38
  gettimeofday, 17
  malloc, 37, 38, 40, 49, 50
  mmap, 26, 49
  prctl, 42
  printf, 34
  pthread_barrier_destroy, 16
  pthread_barrier_init, 16
  pthread_barrier_wait, 16
  pthread_create, 28, 40–42
  pthread_exit, 45
  pthread_mutex, 19
  pthread_ setname np, 42
  ptrace, 36
  sbrk, 50
  wait, 9, 13
Linux Variable
  environ, 8

main, 11, 12, 30, 31, 43
MPI, 3, 5, 8
  mpiexec, 4
MPI process, 8

OpenMP, 3, 5

PIC, 35
PIE, 28, 35
PiP-gdb, 23, 45, 52

PiP-glibc, 42, 44, 45, 47, 48, 52
PiP Command
annul_rpath, 48
pip-check, 21
pip-exec, 3, 4, 8, 21, 22, 24, 49
pip-gdb, 23, 45, 50
pip-mode, 23
pip-pip, 52, 53
pip-unpie, 47
pipcc, 3, 4, 21, 28, 30, 44, 47
--cflags, 21
--lflags, 21
--pipboth, 21
--piproot, 21
--piptask, 21, 28
--which, 21
pipfc, 21, 28, 30, 44, 47
--cflags, 21
--lflags, 21
--pipboth, 21
--piproot, 21
--piptask, 21, 28
--which, 21
piplnlibs, 47
pips, 22, 23, 36, 37, 42
printpipmode, 23
PiP Define Symbol
PIP_CPUCORE_ABS, 31
PIP_CPUCORE_ASIS, 8, 31
PIP_MODE_PROCESS, 29
PIP_MODE_PTHREAD, 29
PIP_MODE_THREAD, 29
PIP_PIPID_ANY, 9
PIP_PIPID_ROOT, 8, 10
PiP Environment Variable
PIP_GDB_COMMAND, 37
PIP_GDB_PATH, 36
PIP_GDB_SIGNALS, 37
PIP_MODE, 29, 40, 41
PIP_SHOW_MAPS, 36, 37
PIP_SHOW_PIPS, 36, 37
PIP_STACKSIZE, 30, 31
PIP_STOP_ON_START, 36
pipcc
CC, 21
pipfc
FC, 21
PiP Function
pip_spawn, 24
pip_barrier_fin, 16
pip_barrier_init, 16
pip_barrier_wait, 16, 17, 24
pip_exit, 15, 24
pip_exit(), 29
pip_fin, 9
pip_get_pipid, 47
pip_gettime, 17
pip_glibc_lock, 45
pip_glibc_unlock, 45
pip_init, 8–10, 17, 24, 29
pip_is_shared_fd, 29
pip_is_shared_fd(), 29
pip_is_threaded, 29
pip_is_threaded(), 29
pip_kill(), 29
pip_named_export, 7, 24, 33, 34
pip_named_import, 7, 24, 33, 34
pip_sigmask(), 29
pip_signal_wait(), 29
pip_spawn, 8, 9, 11, 12, 30, 31, 44
pip_spawn_from_func, 12
pip_spawn_from_main, 12
pip_spawn_hook, 31
pip_task_spawn, 11, 12, 24, 30, 31, 44
pip_trywait, 14
pip_trywait_any, 14
pip_wait, 9, 13, 14, 24
pip_wait(), 29
pip_wait_any, 14
pip_yield(), 29
PiP Term
after hook, 33
before hook, 31, 33
ldpip.so, 40, 41
libpip.so, 23, 41

PIPID, 7–10, 14, 24, 35, 42, 44, 51
process:got, 42, 43
process:pipclone, 42, 43
process:preload, 42, 43
process mode, 28, 31, 42
pthread, 43
pthread mode, 28, 31, 42
PiP Type
pip_barrier_t, 17
pip_spawn_hook_t, 31
pip_spawn_program_t, 12, 30