Great Experiences with PiP (Proecss-in-Process)

Atsushi Hori

February 9, 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

List of Figures

List of Tables

Listings

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.

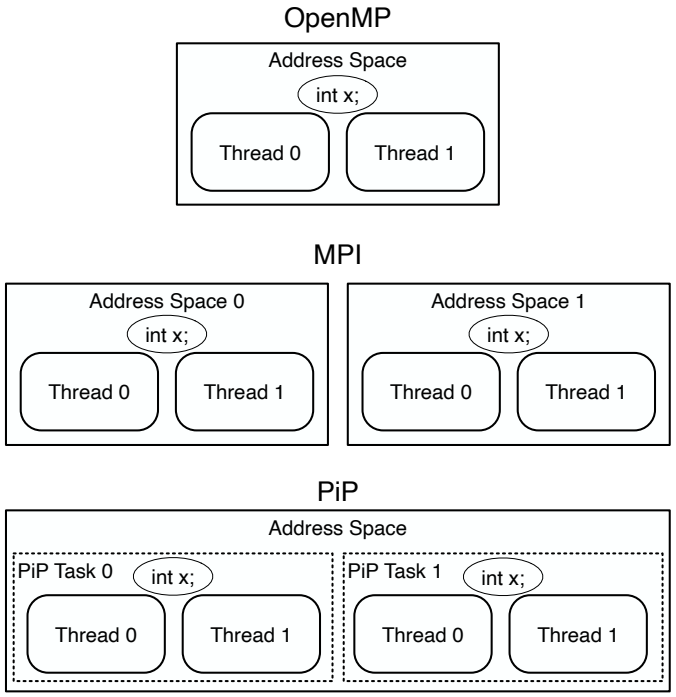

Figure 1.1: Differences of OpenMP, MPI and PiP

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 explain about the timing synchronization among PiP tasks.

1.3.1 Barrier Synchronization

Currently, there is only one synchronization method is supported by the PiP library, it is barrier synchronization. The API of PiP's barrier synchronization is borrowed from the one found in the PThread library. There are three functions in PiP, **pip_barrier_init()**, **pip_barrier_wait()**, and **pip_barrier_fin()**, corresponding to `pthread_barreir_init()`, `pthread_barrier_wait()` and `pthread_barrier_destroy()`, respectively.

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;
}
```

In Listing 1.24, the **pip_init()** function is given a new non-NULL value to the third argument. This is another form of exporting a pointer from the root to spawnd PiP tasks. In this example, the address of the **pip_barrier_t** static variable is passed to children so that the children can synchronize by calling **pip_barrier_wait()**.

To clarify the effect of the barrier synchronization, the synchronization takes place only when the second parameter of the program execution is not given, and then the return values of **pip_gettime()** are shown by PiP tasks. The **pip_gettime()** returns the current value of `gettimeofday()` in double format with the unit of seconds.

The example of running of this program is shown in Listing 1.25. In the first run, the barrier synchronization does not take place and large variance can be seen on the `gettimeofday()` values. In the second run, where the barrier synchronization takes place, and smaller variance can be seen.

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

Users can utilize the synchronization functions on PiP tasks provided by the PThread library. This is simply because PiP tasks share the same address space, just like threads.

1.3.3 pthread_barrier

The same barrier synchronization can also be implemented by using the `pthread_barrier` functions. Listing 1.26 is the program simply replacing **pip_barrier** functions with the `pthread_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

This section will describe on the PiP commands in the PiP package. Some of them are already shown but explained very briefly. In this section, details of PiP commands will be explained.

1.4.1 `pip-man`

This command shows the PiP man pages. Although this is just a simple shell script to run Linux's `man` command with the `MAN_PATH` setting to the PiP man pages (if installed properly), users need not take care about the man path by using this command.

1.4.2 `pipcc` and `pipfc`

As already described in Section 1.1.1, **pipcc** is the compiler script for compiling PiP programs for C and C++ and **pipfc** is for Fortran.

The `--which` option will show you the pass of the actual back-end compiler. Or, users can specify the back-end compiler by setting the environment variable `CC` for **pipcc** or `FC` for **pipfc**.

By default, **pipcc** and **pipfc** compile program to produce the code which can run as a PiP root process and/or a PiP task. Users may specify `--piproot` option for PiP root only program, or `--piptask` option for PiP task only program. Indeed, any PiP program compiled as PiP tasks can run as a PiP root too. Thus, `--piptask` option is equivalent to `--pipboth` (to be both root and task) option.

The actual compile options to be passed to the back-end compiler are shown by specifying the `--cflags` option and the link options are shown by the `--lflags` option. The `--cflags` or `--lflags` disables the actual compiling and/or linking process. All options and parameters not for **pipcc** and those Linux commands cannot run as PiP programs. Additionally, any shell script (shebang) cannot run as a PiP program. As shown in Listing 1.30, the `ls` command is implemented a shell script indeed.

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** program does not guarantee a program to run as a PiP program, even if it tells so.

1.4.3 `pip-exec`

The **pip-exec** command is to invoke PiP tasks derived from one program in the examples so far. However, **pip-exec** can invoke multiple programs and

all PiP tasks derived from those programs share the same address space. To do this, programs are separated by colon (:) (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`

pips is the command to output the list of currently running PiP roots and PiP tasks in the similar way of what the Linux's `ps` command does. Here is the example, running three (3) **pip-exec** each of which execute `a`, `b`, or `c` PiP tasks.

```
$ 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 see, this output looks very similar to the on of the `ps` command. The unfamiliar column titled `PIP` represents if this is a PiP root or PiP task

(first character. 'R' means root, the other numerical digit '0-9' means PiP task. The second character represents PiP **execution mode**, explained in Section 2.2).

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

1.4.5 pip-gdb

pip-gdb is PiP-aware version of gdb (GNU debugger). PiP tasks are implemented as GDB's inferiors. Here is the example of PiP-gdb debugging session.

```
(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 programs must be compiled with the **pipcc** (for C and C++) or **pipfc** (for Fortran) command.

- PiP programs can run as PiP tasks by using the **pip-exec** command.
- PiP programs can run as non-PiP tasks by invoking them as normal programs.
- Unlike the conventional multi-thread model (i.e. OpenMP), static variables in a PiP program are privatized and each PiP task has its own set of the static variables.
- Unlike the conventional multi-process model (i.e. MPI), PiP tasks may share the same address space and PiP tasks can access data owned by the other PiP tasks.

PiP API

- Most PiP functions return error code defined in Linux.
- Every PiP task has a unique **PIPID** per address space.
- PiP root must initialize PiP library by calling **pip_init()**. While child PiP task may or may not call the initialization function.
- PiP root can spawn PiP tasks by calling the **pip_spawn()** or **pip_task_spawn()** function.
- To obtain the address for accessing data of the other PiP tasks, use the **pip_named_export()** and **pip_named_import()** functions.
- The **pip_named_export()** and **pip_named_import()** can be used to synchronize tasks. **pip_barrier_wait()** can also be used for tasks to synchronize.
- The **pip_exit()** function terminates the calling PiP task and PiP root.
- PiP root can wait for the termination of a spawned PiP task, by calling one of the **pip_wait()** function family.

1.6 Myths on PiP

I cannot see the difference between shared memory and what PiP does

The shared memory model enables to access the data owned by the other process. While the POSIX share memory model allows to share only newly allocated memory region, while another shared memory mode provided by XPMEM[3] allows for the other process to access any memory region. Thus both look the same in terms of accessing the data owned by the other.

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

However, the mechanisms of both memory models are quite different. To have a shared memory, one must call a system call to ask OS kernel to have the shared memory. This kind of system calls, modifying the memory mapping, are quite expensive. On the other hand, PiP tasks are mapped in one memory address, and a PiP task can access any data owned by the others once the addresses of the data are known, without calling any expensive system calls. In terms of how memory regions are mapped, I name the way what PiP does **shared address space model**, as oppose to the shared memory model. It should be noted that the shared address space model includes the shared memory model.

If the data to be shared are scattered in an address space and hard to pack in a memory region, or the shared data are being dynamically allocated, then the shared address space model has the advantage.

Sharing an address with multiple programs can be a severe security issue

PiP allows to run programs sharing the same address space. The most important point here is to make information exchange among programs easy and efficient. If there is no information exchange among them, there is no reason to run them with the PiP environment.

Basically, communicating programs share the same fate. Even a most simple case where two programs are connected by using the Linux/Unix pipe, one of the programs dies, the other programs also dies by receiving the `SIGPIPE` signal. Communicating programs agree with others when to communicate and how to communicate. The PiP case is no exception.

Sharing address space makes debugging difficult

It is true if one of the processes in a PiP environment destroy the data owned by the other(s) may lead to a catastrophic result. If this is done maliciously, then this cannot be avoided (see also above). If the destruction is triggered by a software bug, then this might be harder-to-debug than that of multi-process model. There are two points here; 1) the higher possibility of destructing of actual data, not accessing invalid memory region (`SIGSEGV`), and 2) there are multiple execution entities.

The ASLR can be some help for the former point. If ASLR is enabled, then the phenomenons of the bug can vary time to time. The situation of the latter point is almost the same with the multi-thread case.

Anyway, I have no experiences for having bugs based on this situation up until now.

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

You may write programs without having any static variables. However, the functions implemented in Glibc have many static variables. Your runtime system may use some of the Glibc functions. So, in general, it is very hard to write programs not having any static variables.

PiP may consume more memory than the other execution models

The answer is yes, but not that much. Listing 3.4 shows how memory segments of running PiP tasks are mapped in one address space. For example, Glibc (`libc-2.28.so`), Listing 2.4 shows only the memory segments related to Glibc, loaded three (3) times onto memory in this case, running one PiP root and two PiP tasks each of which requires Glibc. Each segment set is a memory map of `libc-2.28.so`. All three segment sets are mapped from the same file, and the amount of consumed memory is the same with having only one set.

In the multi-process model, Each address space of a process has only one `libs-2.28.so` segment set, but another process has also the same memory mapping of Glibc. Thus, roughly speaking, the amount of memory required to run PiP tasks is almost the same with the one of running multiple processes. However, in the multi-thread model, there is only one variable segment shared among threads, regardless to the number of threads. And the amount of memory for running PiP tasks is larger than that of running multiple threads.

There must be some hidden overhead for running PiP programs

So far, it is know that there is one overhead which is larger than the multi-process model. It is address space modification system calls, such as `mmap()` and `brk()`. This is because any modification of an address space must be locked inside of the OS kernel and this lock contention results in larger overhead. This situation is the same with the multi-thread model and the overhead of `mmap()` is larger than the multi-process model but almost the same with the multi-thread model. There is no other known additional overhead in PiP so far.

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 prosedure to spawn a PiP task is (more detailed procedure can be found at Section 3.1.1);

1. create a new name space by calling the Linux's `dlmopen()` function,
2. create a PiP task process (or thread) by calling the Linux's `clone()` system call, and
3. jump into the starting function of a user program.

The `dlmopen()` function can create a new name space, unlike `dlopen()`. Here, the *name space* is the global symbol names (functions and global variables) to be resolved at loading a program. By creating a new name space, functions and variables can be privatized from the other PiP tasks.

The order of calling the `dlmopen()` and `clone()` is very important. At first, I tried to call them in the order of calling `clone()` followed by `dlmopen()`, because this way seemed to be quite natural, however, this does not work at all. This the reason of that only PiP root can spawn PiP tasks and wait for the terminations of PiP tasks.

In some cases (or, in most cases before CentOS/Redhat 8), the loaded address of a program is fixed by default. If this is the case, PiP cannot load multiple programs in the same address space. To enable this, the PiP executables must be compiled as PIE (Position Independent Executable) so that the programs can be loaded at any arbitrary address. All programs to be PiP tasks must be compiled as PIE, i.e., must be compiled with **pipcc** or **pipfc** with the `--piptask` option (or nothing to use the default). Note that PiP root program may not be PIE.

By running the loaded program having a new name space with another thread, PiP task can be created. Unfortunately, things are not that simple. There are many issues coming from Glibc. The next section will describe these issues.

2.2 Execution Mode

PiP library is designed to run on Linux. As described in Section 2.1, it heavily depends on the `dlmopen()` and `clone()`. Especially, the `clone()` is called with a rare combination of `CLONE` flags. There are many Linux variants and some of them do not support such a `CLONE` flag combination (for example, McKernel). To run PiP on such environment, there are two PiP execution modes, one for calling `clone()` with the special flag combination and another for calling `pthread_create()` (using the normal flag combination) to spawn a PiP task. The former is called **process mode** and latter is called **pthread mode**. In either mode, the PiP's basic nature, sharing address space and variable privatization are preserved.

2.2.1 Differences Between Two Modes

The difference of the PiP **execution mode** ends up with the difference of the `clone()` flag combination. Unlike the **pthread mode**, the `CLONE` flags of `CLONE_FS`, `CLONE_FILES`, `CLONE_SIGHAND` and `CLONE_THREAD` are reset, `CLONE_VM` and `CLONE_SYSVSEM` is set.

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

Table 2.1 shows the major differences between the two modes. There are many other differences, though, PiP library provides mode-agnostic functions so that users can write PiP programs without care of the mode differences.

There are also predicate functions for users to know the current mode listed below;

The meaning of **pip_is_threaded()** and **pip_is_shared_fd()** are the same in the current implementation. The reason to have those functions is that there might be the case where those two may have different meanings.

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

Table 2.3: Execution Mode Predicates

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

2.2.2 How to Specify Execution Mode

The execution mode can be specified when to call **pip_init()** and/or setting the **PIP_MODE** environment variable at run-time. Below is the function prototype of the **pip_init()**. The first three arguments are already described up until now.

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

The possible values of the last `opts` argument are one of **PIP_MODE_PROCESS**, **PIP_MODE_THREAD**, `or`ing the both, and zero. The value of zero is equal to `PIP_MODE_PROCESS|PIP_MODE_PTHREAD`. As for the **PIP_MODE** environment, it can be a string of "`process`" or "pthread." When the `opts` value is zero or the value of `or`ing the both, then the value of **PIP_MODE** environment variable is checked. If the environment is not set, then the PiP library chooses an appropriate one. The `opts` value and the environment value cannot not contradict with each other.

2.3 Spawning Tasks - Advanced

In this section, other features, not described so far, of **pip_spawn()** and **pip_task_spawn()** will be explained. For convenience, the function prototypes of these functions are shown below;

```
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]
```

The first argument of **pip_task_spawn()** function is already described in Section 1.2.1. This is structure is to pack the first three arguments of **pip_spawn()**.

2.3.1 Start Function

As already shown in Listing 1.2.1, these PiP spawn functions eventually jumps into the start function (`main()` or user specified one). To enable this, PiP needs to know the address of the start function. One of the following two conditions must be met here;

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

As for the `main()` function, the **pipcc** and **pipfc** compile programs with the `-rdynamic` option to make the symbol global. As for the user-defined local symbol, PiP library read the executable file to spawn and tries to find the starting function by using the ELF information. Unfortunately, the local symbol information is lost if stripped, and PiP fails to find the starting function.

2.3.2 Stack Size

The stack size of spawned PiP tasks can be set by the **PIP_STACKSIZE** environment variable. Like the `OMP_STACKSIZE` environment defined by OpenMP, its value may be suffixed by "`T`," "`G`," "`M`," "`K`," or "`B`" representing the *TiB*, *GiB*, *MiB*, *KiB*, or*Byte* unit, respectively. If no suffix is present, *KiB* is assumed. Unless **PIP_STACKSIZE** is specified, the environment variable `KMP_STACKSIZE`, `GOMP_STACKSIZE`, or `OMP_STACKSIZE` is also effective with the priority in this order. The `KMP_STACKSIZE`, `GOMP_STACKSIZE`, and `OMP_STACKSIZE` also affects the size of OpenMP threads, however, **PIP_STACKSIZE** only affects the stack size of PiP tasks.

2.3.3 CPU Core Binding

The `coreno` argument is to bind the spawned PiP task to the specified CPU core. By default, this is the Nth core number. If users want to specify the absolute core number, then the absolute core number should be `OR`ed with the **PIP_CPUCORE_ABS** flag. If the core numbers are consecutive, specifying this flag may not affect the core number specification. This difference is only seen on some CPU architectures with non-contiguous core numbers (e.g., Fujitsu A64FX). The `coreno` argument can be **PIP_CPUCORE_ASIS** to bind to the same CPU cores as of the root process calling the spawn function.

2.3.4 File Descriptors and Spawn Hooks

In the **process mode**, file descriptors of the root process are duplicated and passed to the spawned child in the same way of what `fork()` does. In the **pthread mode**, files descriptors are simply shared among PiP root and PiP tasks.

If the close-on-exec flag of a file descriptor owned by the root process is set in **process mode**, then the file descriptor is closed after calling the **before hook** described above (if any), and then jump into the start function.

The last argument of **pip_task_spawn()** is the structure packing the last three arguments of **pip_spawn()**. The **pip_spawn_hook_t** structure can be set by calling **pip_spawn_hook()** function. 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* );
```

The `before` function in this structure is called when a PiP task is created and before calling the start function (e.g., `main()`). And the `after` function is called when the PiP task is about to terminate. Both functions are called with the argument specified by the `hookarg` to pass any arbitrary data.

In general, a new process is created by calling `fork()` and `execve()` in Linux/Unix. Here, file descriptors owned by parent process are passed to the created child. In many cases, those file descriptors are closed or duplicated and some other settings take place between the calls of `fork()` and `execve()`. In PiP, however, the task is created by only one function and there is no chance to do the same settings with the ones of using `fork&exec`. These hook functions are provided for this purpose. Here is an example 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 example, FD1 of the root process is duplicated to FD10 by the **before hook**. Then the spawned task write a message via FD10. Finally, the FD10 is closed by the **after hook** (Listing 3.3). Note that the execution of those hook functions are called by the root.

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 explaining the rest of the arguments, readers should know about the execution context under PiP. The execution context can be defined as the state of CPU, i.e., contents of hardware registers. On PiP, this definition may not be enough. Let us have an example. Suppose that the same program runs as two PiP tasks and this program has a function `foo()`. By passing the function pointer, by using the **pip_named_export()** and **pip_named_import()**, one of the PiP task can call the function of the other PiP task. Additionally, this function accesses a static variable, say `var`. If task A calls function `foo` of task B, then the called function accesses the variable owned by task B, not A (Listing 2.3 and 2.4).

Listing 2.3: Function Call 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;
}
```

In this example program, sorry, this may go off the side road, **pip_named_export()** and **pip_named_import()** are called differently than before. The second argument of **pip_named_export()** and the third argument of **pip_named_import()** are actually a format string, just like `printf()`, followed by argument(s) needed 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
$
```

Thus, the execution context in PiP environment might be different from the one in common sense and some times its behavior becomes very subtle. In the PiP library, this happens quite often and makes debugging quite difficult.

Further, the association of static variables and function addresses heavily depends on the CPU architecture and how PIE is implemented. The above description is true on `x86_64` and `AArch64`, however, not true on `x86_32`. Thus, it is not recommended to do this.

Rationale

Some readers may wonder why this happens. Let me explain this. This trick is hidden in the address map. Listing 2.4 shows a part of address map running three tasks, focusing 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
  ...
```

There are three sets of Glibc segments. The static variables are located on the last (readable and writable) segment of each set. A static variable is accessed by an instruction using the offset from the instruction (program counter relative addressing mode) to the variable. Thus, the gap size between the code segment (top of the set) and variable segment (bottom of the set) is important to make all offsets constant and thus all gap sizes must

be the same. In this way, variables and instructions are associated in PIE[1], and PIE programs and shared objects compiled with the PIC option can be loaded at any locations.

Unfortunately, this addressing mode is not supported by all CPU architectures[2]. For example, `x86_32` does not. On this architecture, one general purpose register is sacrificed to point the variable segment, resulting performance degradation by loosing one general purpose register. And the program shown in Listing 2.3 exhibits differently.

2.5 Debugging Support

Some environment variable settings may help debugging PiP programs.

2.5.1 PIP_STOP_ON_START

This environment variable is to stop (by sending `SIGSTOP` to spawned PiP task just before calling the **before hook** (Section 2.3.4), or jumping to the starting function if before hook is not given. The value of this environment must meet the following format;

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

The optional ⟨`script`⟩ is a shell script to be executed on the suspension, and ⟨`PIPID`⟩ is the **PIPID** to be suspended. If ⟨`PIPID`⟩ is -1, then all spawned PiP tasks will be stopped. The ⟨`script`⟩ is invoked with three parameters; PID and **PIPID** of the stopped PiP task, followed by a path to the program of the task. Do not forget to set the *executable* bit on this ⟨`script-file`⟩ file.

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}
```

Listing 2.5 shows an example of the script for the **PIP_STOP_ON_START**. Here, **pips** command is invoked instead of some debugging command[3]. Note that the target task is already stopped by delivering the

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

[2] Listing 2.4 is obtained by running the program on an `x86_64` CPU.

[3] All examples are executed on a Docker environment but `ptrace` (and other commands using `ptrace`) was unable to run in this example even withthe `--cap-add=SYS_PTRACE` Docker option (I confirmed `gdb` worked). So **pips** was used instead in this example.

SIGSTOP signal. Somehow you have to explicitly deliver the SIGCONT signal to the task if you want to resume the task. Listing 2.6 shows the result of **PIP_STOP_ON_START** execution with this script file.

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

This environment variable **PIP_GDB_SIGNALS** is to set the signals to trigger some actions by specifying the **PIP_SHOW_MAPS** and **PIP_SHOW_PIPS**, followed by the PiP-gdb invocation. The value of this environment is as follows;

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

The possible ⟨SIGNAME⟩ vale are listed below;

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

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

Here, ‘‘ALL’’ means all signals list in this table. Each signal name in this table can be concatenated by using the plus (+) and/or minus (-) symbols. For example, “ALL-SIGUSR1” indicates the all signals excluding SIGUSR1. “SIGUSR1+SIGUSR2+SIGINT” represents SIGUSR1, SIGUSR2 and SIGINT.

2.5.3 PIP_SHOW_MAPS

If **PIP_SHOW_MAPS** environment is set to "on" and the a signal specified by the **PIP_GDB_SIGNALS** is delivered, then the address map (Listing 3.4, for example) will be shown.

2.5.4 PIP_SHOW_PIPS

If **PIP_SHOW_PIPS** environment is set to "on" and the a signal specified by the **PIP_GDB_SIGNALS** is delivered, then **pips** command (Section 1.4.4) is invoked to show the status of the other PiP tasks in the same address space.

2.5.5 PIP_GDB_PATH and PIP_GDB_COMMAND

When **PIP_GDB_PATH** is set to the path to **pip-gdb** a signal specified by the **PIP_GDB_SIGNALS** is delivered, then PiP gdb (Section 1.4.5) will be invoked. If the value of the **PIP_GDB_COMMAND** environment is set to a valid filename and if the filename contains some GDB commands, then PiP-gdb will be invoked to work with this command file.

2.6 Malloc routines

Suppose that we are making a producer-consumer style program using PiP, a PiP task is a producer and another PiP task is a consumer. Unlike the conventional process model, there is no need of calling IPC (Inter Process Communication) system call in PiP. All we have to do is just passing pointers pointing data to be passed from the producer to the consumer.

Here, an issue arises. If the passing data is allocated by a `malloc()` routine, then the passed data is `free()`ed by the consumer. As described so far, each PiP task has its own `malloc()` and `free()` routines associated with static variables holding and maintaining a memory pool. The consumer receives the data allocated from the memory pool of the producer and tries to `free()` it when it becomes unnecessary. However, the `free()` routine on the consumer has no knowledge about the producer-allocated memory region and fails (Figure 2.1). I named this situation *cross-malloc-free*.

I tried this by using the `malloc` routines provided Glibc and I found that this works in most cases, not always. I do not know why this works (again, **in most cases**) with the Glibc `malloc` routines, but I believe this situation must be avoided.

To deal with this, PiP library wraps `malloc` routines as shown in Table 3.3. When a memory region is allocated, the `malloc` wrapper function embeds the information who allocates that region. When this region is to be `free()`ed, the `free()` wrapper function connects the region to the freeing

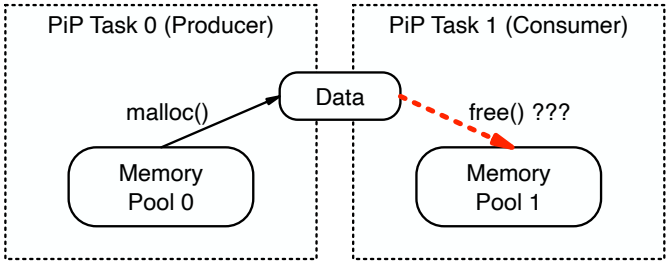

Figure 2.1: Cross-Malloc-Free Issue

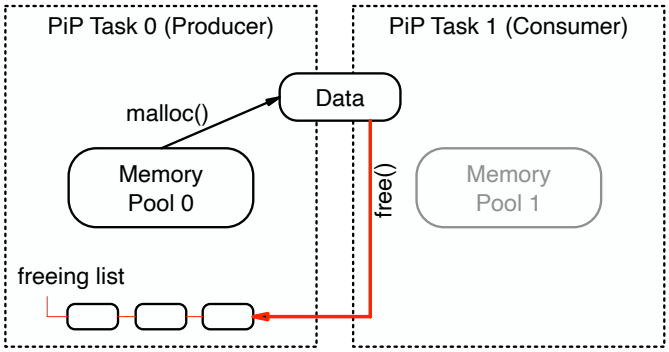

Figure 2.2: Cross-Malloc-Free with Freeing List

list of the task allocating the region. The regions in the freeing list are eventually `free()`ed when one of the `malloc` wrapper functions (Figure 2.2).

2.7 XPMEM

As described in Section 1.6, `XPMEM` is known to provide a shared memory model which is more convenient that the POSIX shared memory. Again, the shared address space which PiP provides includes the shared memory model which `XPMEM` and POSIX shared memory provide. Thus, the same functionalities of `XPMEM` can also be implemented by using PiP.

The PiP library provides the same functions which are provides by `XPMEM`. Those who have programs using `XPMEM` can easily switch to using PiP. By using PiP, there is no need of installing `XPMEM` kernel module. Most importantly, `XPMEM` functions provided by PiP work much faster than those of `XPMEM`. This is because no system call involved to map memory segment of the other process(es) in PiP. Indeed, most `XPMEM` functions almost do nothing since the other processes are already mapped from the beginning in PiP.

Chapter 3

PiP Internals

3.1 PiP Implementation

3.1.1 Spawning Tasks

Before PiP version 2.4, PiP tasks were created with the procedure as follows;

1. The spawned 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.

From PiP version 2.4, the wrapper functions listed in Table 3.3 were introduced. When implementing the wrapper functions, I noticed that wrapping the `dlsym()` is almost impossible.

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()`?

To solve this issue, I implemented another program, so called **ldpip.so** to load the PiP library and user program. here is the details of new spawning process;

1. Load **ldpip.so** 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 **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.

At the time of loading **ldpip.so**, no wrapper functions are defined in this program and obtaining the Glibc function addresses is easy, just referencing them. After loading the **libpip.so** and jumping into a function defined in **libpip.so** where the wrapping functions are defined, the Glibc functions to be wrapped are now wrapped by using the function table created by **ldpip.so**[1].

The Glibc initialization[2] must be done with the execution context (Section 2.4) of the spawned PiP task. In the older version of PiP library, this was done by; 1) calling `dlsym()` to the loaded handle, returned by `dl[m]open()`, to obtain the initialization function and then 2) call the function. In the new implementation, the initialization was done by simply calling the initialization function from the **ldpip.so** where the execution context is the same with that of PiP task.

Thus, by introducing PiP loader program (**ldpip.so**), things can go in a simpler way.

3.1.2 Calling `clone()` System Call

As described in Section refsec:spawn-details, PiP library calls the `clone()` system call with a special flag combination. The `clone()` system call has many arguments and some of them are hard to implement, I decided to wrap the `clone()` system call to modify only the flag setting.

One issue to wrap the `clone()` system call is that the `clone()` is called not only the PiP library, but also some other libraries (e.g., PThread librray). A simple function wrapping cannot handle both situations where the flags

[1] If actual dynamic linking would be done in the order of `dl[m]open()`, then the wrapping functions in `libpip.so` would not work as described here. As long as I checked, the Glibc (`libc.so`) is at the last of the search order of `ld-linux.so`, and this works.

[2] Calling `__ctype_init()`

needs to be changed when it is called by the PiP library but it should not change the flag when called by some other libraries.

To solve this issue and protect the `clone()` system call from being called from PiP library and from another simultaneously, a special locking mechanism was implemented by using the `test-and-set` atomic instruction. When the PiP library is about to call `pthread_create()` call, which eventually calls the `clone()` system call, it locks by using the `test-and-set` instruction with the value of current thread ID (TID). 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 succeeds, then it is called by some other library. In the former case, the original `clone()` is called with the modified flags. In the latter case, the `clone()` is called with the same argument with the wrapper function call. Needless to say, the lock is unlocked after returning from the original `clone()` system call.

3.1.3 Execution Mode in Details

There are two sub-modes in the PiP's **process mode**, **process:preload** and **process:pipclone**[3]. The **process:preload** mode is implemented by wrapping the `clone()` function described above and the **process:pipclone** is implemented to have another `pthread_create()` like function implemented in the patched-glibc (Section 3.2.2). If PiP library is configured to use the patched-glibc, then **process:pipclone** is taken, otherwise **process:preload** is taken.

3.1.4 Name of PiP Tasks

Some readers may wonder how the **pips** command (Section 1.4.4) can distinguish the PiP tasks and the other normal processes and/or threads. In Linux, each process and thread can have a name, which can be seen by the `top` command in the `COMMAND` column. The PiP library sets the command name by calling the `prctl()` system call (in **process mode** or `pthread_setname_np()` call (in **pthread mode**. The PiP library uses the first two characters of the name (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**

[3]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	

The name may have up to 16 characters and PiP occupies the first two characters The remaining 14 characters are used for representing original command name. The abbreviation column in Table 3.2 shows the characters used by the **pip-mode** command (Section 1.4.6) to specify the PiP execution mode. The **pips** command can now distinguish the normal processes or threads from the PiP roots and PiP tasks by using those first 2 characters of the command.

3.2 Issues related to Linux Kernel, Glibc and Tools

This section will explain about the issues when implementing PiP.

3.2.1 Loading a Program

Before going into the details about the Glibc issues when implementing PiP, readers should understand how a program is loaded into memory. This subsection describes only about the program loading procedure of Linux, apart from PiP implementation.

When the Linux's `execve()` system call to run a program, the Linux kernel open and read the executable file, searching the ELF section named "`.interp`."

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]
$
```

Listing 3.1 shows the value of the `.interp` section, `/lib64/ld-linux-x86-64.so.2`. The Linux kernel invokes the loader specified by the `.interp` section and asks the loader to load a program specified by the `execve()` parameter. Then the loader load the program and additionally load and link the shared libraries required to run the program. Once everything is loaded, the loader jumps into the starting function defined in Glibc to initialize Glibc and finally user-defined `main()` function is called.

The program loader, often simply called `ld-linux.so`, is loaded once per address space and kept in memory until the end of the process (see also Listing 3.4). This is responsible for any loading process by resolving the

external symbol references. The Glibc functions defined in the `libdl.so` (`-ldl`), such as `dlopen()`, `dlmopen()`, `dlsym()` and so on, are just API and their functional bodies exist in this program loader.

3.2.2 Glibc

PiP provides a new execution model which cannot be categorized into neither the process model nor the thread model. In this new model, although its name is not yet given, tasks share the same address space like the thread model, but maintaining the variable privatization like the process model. This execution model is novel and not yet recognized by most of the tool chains provided by Linux and others. Indeed, the most of the time to develop PiP was devoted to find niches in Glibc.

PiP Task is Unable to Spawn PiP Task

As described in Section 2.1, the order of calling `dlmopen()` and `clone()` is important. This restriction also means that a PiP task cannot spawn a PiP task as a child of spawning PiP task because this breaks the restriction. Thus, the current PiP implementation inhibits for a PiP task to call **pip_spawn()**.

Recycling PiP Tasks

As far as I tested, the resources; name space, loaded PIE program, and shared libraries required by the PIE program, are not released by calling the `dlclose()`. I believe this issue can be fixed by patching the Gilbc, however, I decided no to do so. Thus, once a PiP task is created, then the **PIPID** of the task will not be recycled even the PiP task terminates. The reason of my decision will also be discussed in Section 3.3.1.

Number of name spaces

The number of names spaces which the `dlmopen()` can create is hard-coded as 16. Considering PiP tasks run in parallel and the number of CPU cores nowadays, this number of 16 is apparently too small. The PiP package provides PiP-glibc where the number of name spaces is increased, up to 300 PiP tasks[4].

The name space table resides in the `ld-linux.so` and this means that the `.interp` ELF section of PiP programs must be changed so that the program is loaded by the new `ld-linux.so`. This can be done by specifying `--dynamic-linker` option of the GNU linker and the **pipcc** and **pipfc** do this.

[4]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

The name space table resides at the top of a structure in `ld-linux.so`. Some Glibc functions refer to the members in this structure directly. This causes another problem. Once the size of the name space table is changed, the addresses of the other members in the same structure are also changes. As described, only one `ld-linux.so` can be loaded in an address space. As a result, all PiP programs sharing the same address must be linked with the same Glibc.

PiP-gdb

The `ld-linux.so` embeds some information for debugging into the loaded program. Unfortunately, I found that this code fragment resides on the pass calling the `ld-linux.so` from the top (by the kernel), not on the pass called from `dlopen()` and `dlmopen()`[5]. The patched PiP-glibc fixed this issue. Thus, the **pip-gdb** command (Section 1.4.5) can only work with the PiP programs linked with the patched PiP-glibc.

Global lock

Most programs are linked with Glibc and PiP programs are no exception. PiP allows to run multiple PiP programs in the same address space. This means that each PiP task has its own Glibc. And the simultaneous calls of some Glibc functions may not work because of a race condition.

To avoid this condition, PiP library provides the functions, **pip_glibc_lock()** and **pip_glibc_unlock()**, to serialize the Gilbc function calls. The following Glibc functions are wrapped by PiP library to introduce the lock and users do not have to care 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`

The functions `pthread_exit()` and below in this table have another reason to have function wrappers. The wrapping reason of `pthread_exit()` will be explained in the Section 2.2 and the reason of wrapping `malloc` routines will be explained in Section 2.6.

[5] I guess the loaded code by using `dlopen()` or `dlmopen()` cannot be debugged.

These listed functions may not be complete. There can be a case where some other Glibc functions may suffer from the race condition. This problem can be avoided by introducing the above locking functions. This lock can be used recursively and users can avoid deadlock situation easily.

Constructors and Destructors

The constructors and destructors are used in C++ programs. Constructors and destructors are list of functions. Generally, constructor functions are called just before the program begins, and destructors functions are called when the program is about to exit.

In PiP, the behavior of the constructors and destructors is somewhat different. To explain this, I should start explaining how the constructors and destructors are implemented in general. The constructor functions are listed in the `.init_array` section of an ELF file. The destructor functions are listed in the `.fini_array` section. Constructors are called when `ld-linux.so` finishes loading and linking objects. Destructors are called when `dlclose()` is called.

Now back to PiP. Again, constructors are called inside of the call of `dlmopen()` when spawning a PiP task. The `dlmopen()` is called by the PiP root process. Thus, the constructors of a program are called by the root. 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;
}
```

Listing 3.2 is a C++ program having a constructor and destructor. When the constructor of this program is called, the PiP library is not yet initialized, and the **pip_get_pipid()** return an error (`EPERM`). So, the function `pipidstr()` takes care of this situation. Listing 3.3 shows the executoin example of this program. As shown, the PIDs output by the constructors are not the same with the ones 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`

`LD_PRELOAD` only works with PiP root, not PiP tasks. This is because `dlmopen()` simply ignores the `LD_PRELOAD` environment setting.

Shared Objects

Some shared objects, such as GCC related runtime libraries, must be located in the same directory where the `ld-linux.so` does. The **piplnlibs** shell script found in the PiP-glibc package makes symbolic links of the shared objects in the `/lib64` directories to meet with the restriction.

Loading Program by `dlmopen()`

The Glibc in CentOS/RedHat 8 (and possibly newer ones) does not allow to load a program by the `dlmopen()` function[6]. The **pip-unpie** program is to cheat this Glibc restriction. This program is automatically executed by the **pipcc** or **pipfc** when creating a PiP executable, and not to be invoked by users directly.

3.2.3 Glibc `RPATH` Setting

When using Spack[7], it automatically adds `RPATH`s for every program and PiP is no exception. A problem arises when to install the PiP-glibc by using Spack. When the PiP-glibc is built by using Spack, Spack adds the `RPATH` setting to the compiled Glibc, but it is not allowed in CentOS/Redhat 8 to load Glibc with the `RPATH` setting. To avoid this, PiP-gilbc has a program (**annul_rpatha**) to unset the `RPATH` setting of the compiled Glibc.

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
```

[6] Refer https://sourceware.org/bugzilla/show_bug.cgi?id=11754#c15. I tested this situation but I cannot find this problem with PiP.

[7] PIPKW{pip-unpie}program

```
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
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[8].

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.

[8] `ltrace` depends on its version

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