Great Experiences with PiP (Proecss-in-Process)

Atsushi Hori

June 21, 2022

Preface

I retired from my work from April 2022. Since then, I wrote this PiP tutorial. At the same time, I am maintaining PiP library (https://github.com/procinproc).

Since PiP provides a new execution model not yet widely accepted, I am not confident with adapting PiP with the newer Glibc (currently supporting the Glibc attached with CentOS/Redhat 7 and 8). This is because the PiP implementation is heavily depending on the niches of Glibc.

Acknowledgment

The predecessor of PiP is PVAS (Partitioned Virtual Address Space) developed by Akio Shimada. The idea of using PIE was his idea. Pavan Balaji has a great insight and he saw through the possibilities of PVAS at the first time Akio and I had a meeting with him. He also gave me some great ideas when developing PiP. The name of PiP was his recommendation. Min Si devoted her lots of time to help me writing PiP papers. Kaiming Ouyang wrote great papers to improve MPICH performance by using PiP. Yutaka Ishikawa allowed me to devote most of my time for developing PiP. Balazs Gerofi also gave me some useful comments on PiP. Noriyuki Soda helped me a lot while developing PiP and he developed the Github actions for testing PiP.

Finally, I thank to my wife, Fusa Hori, allowing me to devote my time for writing this book.

Contents

Listings

List of Figures

List of Tables

Introduction

This is a book explaining Process-in-Process (PiP in short) library. This somewhat strange name library is to provide a relatively new execution model to have the best of two world; multi-process and multi-thread execution models.

It becomes quite common to have multiple CPU cores in a CPU socket or die, parallel execution environment also becomes very crucial for efficiency. In the multi-process model, where multiple processes run on a socket, a process cannot directly access data owned by the other processes, in spite of the fact accessing the same physical memory device. In most cases, processes exchange information via some form of communication. In my personal understanding, communication is accompanied with some form of data copying, regardless done by software or hardware. Data copying consumes memory, time and power and it must be avoided as much as possible. What if processes can access data owned by the others? In multi-thread model, threads share the static variables and they must be protected from race conditions if threads try to update their contents. What if each thread has its own static variable set?

This is my motivation to develop PiP. The name of *Process-in-Process* may suggests, a process can create another processes inside of the address space of the creating process. This sounds like the multi-thread execution model, however, the name of *process* in PiP means that each created processes have its own static variable set unlike the multi-thread model. Thus, the created processes share the same address space can access data owned by the others while maintaining the privatized static variables. This way can avoid the data copying accompanied by communication.

Basically, multi-process model shares nothing, multi-thread model shares everything, and everything is sharable in the execution model PiP provides.

With my regards to some predecessors, there are some other implementations providing this kind of execution model. However, PiP is quite unique since it is implemented purely at the user-level, no need of new or patched OS kernel, nor having new language processing systems.

I have been working on high performance computing (HPC) and very little knowledge on the other fields. I can only imagine HPC applications. However, I believe that the easy-to-use nature of PiP can be applied to other

fields.

Chapter 1

PiP Basics

Let me start describing the PiP basics for those who are not familiar with PiP; 1) how to run a PiP program, 2) How to write a PiP program, and 3) usage of PiP commands. The explanations in this chapter does not go into details. For more details, refer to the Chapter 2 and/or the other documents (man pages and PDF).

1.1 PiP Tasks

This section will describe how PiP tasks are created in a simple way and how PiP tasks works in the different way from the process (using MPI) and thread (using OpenMP) creations.

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

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

As you can see, this program is exactly the same with the normal C program. If this program is compiled with the **pipcc** command, then this program can run as a normal 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
$ ./a.out
Hello World
```

```
$ pip-exec ./a.out
Hello World
$
```

The **pipcc** command is written as a shell script to call a real C compiler with appropriate options, such as `-I`, `-L` and so on. If the `--silent` option is omitted, then you will see the options how **pipcc** script calls the backend C/C++ compiler.

The **pip-exec** command in this example is to execute an executable file as PiP tasks, not as a normal Linux process.

This example does not show how the hello program behaves differently between the process and PiP task. The next section wil discuss on this point.

1.1.2 Comparing MPI, OpenMP and PiP

To explain the difference between process and PiP task, we slightly modify the "hello world" program as below;

Listing 1.3: Hello World having a static variable

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

Now the "Hello World" program has a static variable `x` and its address is printed out with the "Hello World" message. The **pip-exec** command may take an option to specify the number of PiP tasks to be created and executed in parallel. In the following execution example, the number of three (3) is specified. Additionally, the output of the same `a.out` execution using MPI. 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-omp.c
$ ./a.out
Hello World (&x:0x555555601030)
$ pip-exec -n 3 ./a.out
Hello World (&x:0x7ffff67d9030)
Hello World (&x:0x7ffff50db030)
Hello World (&x:0x7fffef92d030)
$ mpiexec -n 3 ./a.out
Hello World (&x:0x555555601030)
Hello World (&x:0x555555601030)
Hello World (&x:0x555555601030)
```

```
$
```

The first execution of `a.out` shows that the variable `x` is located at the address of `0x555555601030`[1]. This situation is the same with the MPI execution[2]. However, the locations of the variable `x` executed as PiP tasks are all different. This is because PiP tasks share the same address space but MPI does not. Readers may notice that threads also share the same address space and wonder the difference between PiP and OpenMP. The example below is the OpenMP version of the "Hello World" with a static variable.

Listing 1.5: Hello World in OpenMP

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

The execution output of program 1.5 is shown below. Here, the addresses of variable `x` are the same in OpenMP and MPI executions. However, the addresses of the variable with PiP execution are different pairs.

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

```
$ pipcc --silent -fopenmp hello-var-omp.c
$ export OMP_NUM_THREADS=2
$ ./a.out
Hello World (&x:0x555555601038)
Hello World (&x:0x555555601038)
$ pip-exec -n 2 ./a.out
Hello World (&x:0x7ffff67d9038)
Hello World (&x:0x7ffff67d9038)
Hello World (&x:0x7ffff46a2038)
Hello World (&x:0x7ffff46a2038)
$ mpiexec -n 2 ./a.out
Hello World (&x:0x555555601038)
Hello World (&x:0x555555601038)
Hello World (&x:0x555555601038)
Hello World (&x:0x555555601038)
$
```

Figure 1.1 explains these differences. In OpenMP, the OpenMP threads share the same address space and variable `x` is shared among threads. In MPI, each MPI process has its own address space and two (2) threads run in

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

[2]MPICH implementation where each MPI rank has its own address space.

each address space and share the variable in the same MPI process. In PiP, all PiP tasks share the same address space, however, each PiP task has its own variables and thread 0 and 1 share the variable in the same PiP task, but not sharing the variables in the different PiP task (Figure 1.1).

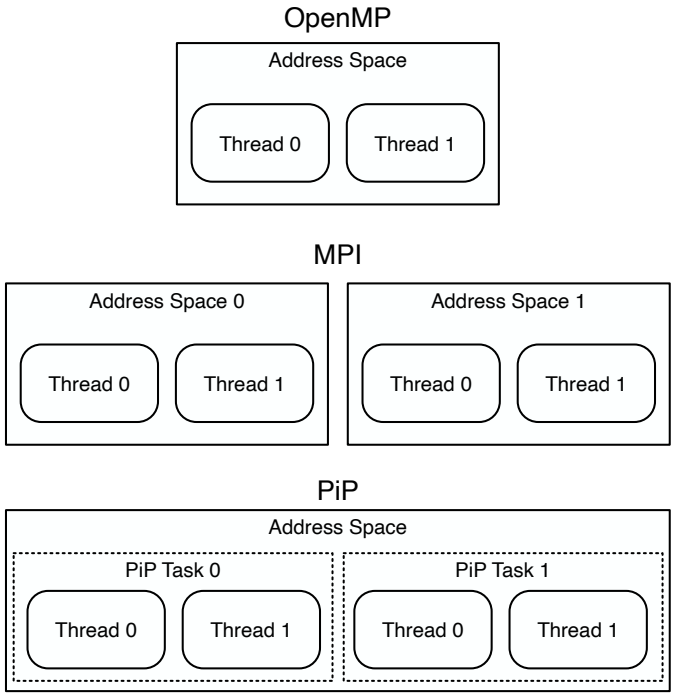

Figure 1.1: Differences of OpenMP, MPI and PiP

In the conventional process model and thread model, static variables are associated with an address space. Thus, each process has its own static variables and threads running on the same address space share the same static variables. In the PiP execution model, each PiP task is guaranteed to have its own static variable set, decoupling from the address space while maintaining the address space sharing. This is called **variable privatization**.

This nature of PiP, privatized variables and sharing an address space, makes it easy to exchange information among PiP tasks while maintaining the independence of each PiP task execution. So far, it is shown that the "Hello World" program can run as PiP tasks in parallel, but this program is so simple and no information exchange among PiP tasks. In the next section, we will show how information can be exchanged among PiP tasks.

1.1.3 Export and Import

Sharing an address space means that data owned by a PiP task can be accessed if the address of the information to be exchanged is known. A PiP

task can publish the address of the data to be shared and the other PiP task(s) can get the published address.

Firstly, each PiP task has **PIPID** to distinguish from the others. A PiP task can export an address so that the other PiP tasks sharing the same address space can import the address by specifying the **PIPID** who exported.

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, a PiP task having **PIPID** of zero (0) export the address of the variable `x` by calling **pip_named_export()** after setting the value of `argv[1]`. The other PiP tasks import the exported address by the PiP task 0 by calling **pip_named_import()** function. Below is an execution result of this program. As shown, the exported value by PiP task 0 can be seen by the other PiP tasks.

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 **pip_named_export()** function publishes an address with the given name. The **pip_named_import()** function blocking-waits until the named address on the specified PiP task by **PIPID**. It is not allowed to export an address having the same name twice or more to update the address, because this leads to a race condition.

Almost all functions provided by the PiP library return an integer value as an error code. The return code of zero (0) means success. This error code is the same with the ones defined by Linux. In the examples so far and hereinafter, the returned code is not checked because of simplicity and readability.

In MPI, it is not allowed to access the data owned by the other processes in the same node. Communication is the only way allowed in MPI[3]. Basically, communication involves some form of data copying (done by software or hardware). Data copying consumes time, power and memory.

1.2 Spawning PiP Tasks and Waiting Terminations

The **pip-exec** command spawns PiP tasks. The process which spawns PiP tasks is called **(PiP) root** process. The **pip-exec** process is a PiP root process. PiP tasks spawned by the root process are mapped and executed in the address space of the root. In this chapter, how to spawn PiP tasks will be explained.

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 **pip_init()** may look strange because this function behaves differently depending on if it is called from a PiP root or PiP task. The first argument is output returning **PIPID** of the calling task. The second input arguments is to specify the maximum number of spawning PiP tasks. This second argument becomes output if this is called by a PiP task, returning the number specified by the root. 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`).

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

[3]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.

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

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 = i;
    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. 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 both or either PiP root and PiP task. Listing 1.12 shows an example of combining the programs of Listing 1.9 and 1.10. We hope you can understand the strange behavior of **pip_init()** function. The PiP root process also acts like a PiP task. It has a special PIPID, `PIP_PIPID_ROOT`. Listing 1.13 shows the example 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`

PiP tasks start from the `main()` function in the examples so far. PiP allows for PiP tasks to start user-defined function other than `main()`. In this case, use the **pip_task_spawn()** function instead of calling the **pip_spawn()** function.

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;
}
```

Listing 1.14 is the program of this example. To decrease the number of arguments to spawn a PiP task, the **pip_spawn_program_t** structure is defined. This structure holds all information for spawning a program, including path to executable file, function name, and so on. To hide the details of the structure, **pip_spawn_from_func()** function is also defined to set these information. The user-defined function must have one argument (`void*`) and return an integer value which is the same as the return value from the `main()` function.

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 firstly introduced (from version 1). After then, I noticed users can start PiP tasks other than main, and the **pip_task_spawn()** function was introduced (from version 2 or later). The **pip_spawn_program_t** structure must be set by calling the **pip_spawn_from_main()** function when starting from the `main()` function. Listing 1.16 is the program rewritten version of Listing 1.12 by using the **pip_task_spawn()** and **pip_spawn_from_main()**.

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 noticed, the **pip_wait()** is the function to wait for terminations of the spawned PiP tasks. The **pip_wait()** function acts like the Linux's `wait()` function. In many cases, Linux's `wait()` function works with PiP tasks, but there is a certain case it does not. So, it is recommended for users to use **pip_wait()** function.

The argument of the **pip_wait()** is the pointer to an integer variable, the same with the Linux's `wait()` call. The returned integer can be examined by using the Linux's `WIFEXITED`, `WIFSIGNALED`, `WEXITSTATUS`, `WIFSIGNALED`, and `WTERMISIG` macros.

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

PiP tasks and root can terminate their executions by calling **pip_exit()** function. This function acts like the Linux's `exit()` function. As described above, it is recommended to use **pip_exit()** instead of `exit()`, because the Linux's `exit()` function works in most cases, however, there is a case it does not. Listing 1.22 and 1.23 show the example showing how **pip_exit()** works.

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 Synchronizing Timing 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 1655800230.082913 [S]
PIPID:1 1655800230.089411 [S]
PIPID:2 1655800230.096152 [S]
PIPID:3 1655800230.102750 [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 1655800230.137033 [S]
PIPID:0 1655800230.137029 [S]
PIPID:2 1655800230.137103 [S]
PIPID:1 1655800230.137097 [S]
PiP task (PIPID:3) done
PiP task (PIPID:0) done
PiP task (PIPID:2) done
PiP task (PIPID:1) 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 1655800230.429683 [S]
PIPID:1 1655800230.434080 [S]
```

```
PIPID:2 1655800230.439602 [S]
PIPID:3 1655800230.445089 [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:1 1655800230.483778 [S]
PIPID:0 1655800230.483770 [S]
PIPID:3 1655800230.483753 [S]
PIPID:2 1655800230.483809 [S]
PiP task (PIPID:1) done
PiP task (PIPID:0) done
PiP task (PIPID:3) 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;
void increment( sync_t *syncp ) {
  int tmp;
  pthread_mutex_lock( &syncp->mutex );
  tmp = syncp->count;
  usleep( 10 );
  syncp->count = tmp + 1;
  pthread_mutex_unlock( &syncp->mutex );
}
int main() {
  int pipid, ntasks, i;
  sync_t *syncp;
  pip_get_pipid( &pipid );
  pip_get_ntasks( &ntasks );
  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\n", syncp->count );
  return 0;
}
```

Listing 1.29: Pthread Mutex - Execution

```
$ pip-exec -n 10 ./pthread-mutex
count=10000
$
```

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, 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 2.3).

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 2.3).

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

```
$ export PIPLIBDIR=${PIPDIR}/lib
$ ${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/install/glibc/lib
ld-linux:       /home/ahori/install/glibc/lib/ld-2.28.so
Commit Hash:    581c0c7618a6bb62a4c155dc826da5698ec34480
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 Some Myths on PiP

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.

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;

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.
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. 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 Issues related to Linux Kernel, Glibc and Tools

This section will explain about the issues when implementing PiP.

2.2.1 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 2.1: `.interp` Section of the `ps` command

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

Listing 2.1 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. 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 the program loader.

2.2.2 PiP-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.

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[1].

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.

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 a tiny 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()`[2]. 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-gkibc.

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.

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

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

Table 2.1: 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 of this have another reason to have function wrapper. The wrapping reason of `pthread_exit()` will be explained in the Section 2.3 and the reason of wrapping malloc routines will be explained in Section 2.7.

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 2.2: 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 2.2 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 2.3 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 2.3: Constructors and Destructors - Execution

```
$ ./hello
[R] PID:3757 Hello
[R] PID:3757 MAIN
[R] PID:3757 Bye
$ pip-exec -n 2 ./hello
[R] PID:3758 Hello
[0] PID:3759 MAIN
[0] PID:3759 Bye
[R] PID:3758 Hello
[1] PID:3760 MAIN
[1] PID:3760 Bye
$
```

`pip-unpie` program

The Glibc in CentOS/RedHat 8 (and possibly newer ones) does not allow to load a program by the `dlmopen()` function[3]. 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.

2.2.3 Linux Kernel

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 2.4: A Memory Map Example

```
00400000-00402000 r-xp 00000000 00:71 73004142                           /PiP/bin/pip-exec
00601000-00602000 r--p 00001000 00:71 73004142                           /PiP/bin/pip-exec
00602000-00603000 rw-p 00002000 00:71 73004142                           /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
7ffff0000000-7ffff0021000 rw-p 00000000 00:00 0
7ffff0021000-7ffff4000000 ---p 00000000 00:00 0
7ffff46d9000-7ffff46da000 ---p 00000000 00:00 0
7ffff46da000-7ffff4eda000 rw-p 00000000 00:00 0
7ffff4eda000-7ffff4edb000 r-xp 00000000 00:71 73011107   /PiP/example/a.out
7ffff4edb000-7ffff50da000 ---p 00001000 00:71 73011107   /PiP/example/a.out
7ffff50da000-7ffff50db000 r--p 00000000 00:71 73011107   /PiP/example/a.out
7ffff50db000-7ffff50dc000 rw-p 00001000 00:71 73011107   /PiP/example/a.out
7ffff50dc000-7ffff50f6000 r-xp 00000000 00:71 73004125   /PiP/lib/libpip.so.0
7ffff50f6000-7ffff52f6000 ---p 0001a000 00:71 73004125   /PiP/lib/libpip.so.0
7ffff52f6000-7ffff52f7000 r--p 0001a000 00:71 73004125   /PiP/lib/libpip.so.0
7ffff52f7000-7ffff52f8000 rw-p 0001b000 00:71 73004125   /PiP/lib/libpip.so.0
7ffff52f8000-7ffff53f8000 rw-p 00000000 00:00 0
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
7ffff57aa000-7ffff57ae000 rw-p 00000000 00:00 0
7ffff57ae000-7ffff57c5000 r-xp 00000000 fe:01 3050215    /lib64/libpthread-2.28.so
7ffff57c5000-7ffff59c4000 ---p 00017000 fe:01 3050215    /lib64/libpthread-2.28.so
7ffff59c4000-7ffff59c5000 r--p 00016000 fe:01 3050215    /lib64/libpthread-2.28.so
7ffff59c5000-7ffff59c6000 rw-p 00017000 fe:01 3050215    /lib64/libpthread-2.28.so
7ffff59c6000-7ffff59ca000 rw-p 00000000 00:00 0
7ffff59ca000-7ffff59cc000 r-xp 00000000 fe:01 3049753    /lib64/libdl-2.28.so
7ffff59cc000-7ffff5bcc000 ---p 00002000 fe:01 3049753    /lib64/libdl-2.28.so
7ffff5bcc000-7ffff5bcd000 r--p 00002000 fe:01 3049753    /lib64/libdl-2.28.so
7ffff5bcd000-7ffff5bce000 rw-p 00003000 fe:01 3049753    /lib64/libdl-2.28.so
7ffff5bce000-7ffff5bd6000 r-xp 00000000 00:71 73004126   /PiP/lib/ldpip.so
7ffff5bd6000-7ffff5dd5000 ---p 00008000 00:71 73004126   /PiP/lib/ldpip.so
7ffff5dd5000-7ffff5dd6000 r--p 00007000 00:71 73004126   /PiP/lib/ldpip.so
7ffff5dd6000-7ffff5dd7000 rw-p 00008000 00:71 73004126   /PiP/lib/ldpip.so
7ffff5dd7000-7ffff5dd8000 ---p 00000000 00:00 0
7ffff5dd8000-7ffff65d8000 rw-p 00000000 00:00 0
7ffff65d8000-7ffff65d9000 r-xp 00000000 00:71 73011107   /PiP/example/a.out
7ffff65d9000-7ffff67d8000 ---p 00001000 00:71 73011107   /PiP/example/a.out
7ffff67d8000-7ffff67d9000 r--p 00000000 00:71 73011107   /PiP/example/a.out
7ffff67d9000-7ffff67da000 rw-p 00001000 00:71 73011107   /PiP/example/a.out
7ffff67da000-7ffff67f4000 r-xp 00000000 00:71 73004125   /PiP/lib/libpip.so.0
7ffff67f4000-7ffff69f4000 ---p 0001a000 00:71 73004125   /PiP/lib/libpip.so.0
7ffff69f4000-7ffff69f5000 r--p 0001a000 00:71 73004125   /PiP/lib/libpip.so.0
7ffff69f5000-7ffff69f6000 rw-p 0001b000 00:71 73004125   /PiP/lib/libpip.so.0
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
7ffff6da8000-7ffff6dac000 rw-p 00000000 00:00 0
7ffff6dac000-7ffff6dc3000 r-xp 00000000 fe:01 3050215    /lib64/libpthread-2.28.so
```

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

```
7ffff6dc3000-7ffff6fc2000 ---p 00017000 fe:01 3050215    /lib64/libpthread-2.28.so
7ffff6fc2000-7ffff6fc3000 r--p 00016000 fe:01 3050215    /lib64/libpthread-2.28.so
7ffff6fc3000-7ffff6fc4000 rw-p 00017000 fe:01 3050215    /lib64/libpthread-2.28.so
7ffff6fc4000-7ffff6fc8000 rw-p 00000000 00:00 0
7ffff6fc8000-7ffff6fca000 r-xp 00000000 fe:01 3049753    /lib64/libdl-2.28.so
7ffff6fca000-7ffff71ca000 ---p 00002000 fe:01 3049753    /lib64/libdl-2.28.so
7ffff71ca000-7ffff71cb000 r--p 00002000 fe:01 3049753    /lib64/libdl-2.28.so
7ffff71cb000-7ffff71cc000 rw-p 00003000 fe:01 3049753    /lib64/libdl-2.28.so
7ffff71cc000-7ffff71d4000 r-xp 00000000 00:71 73004126   /PiP/lib/ldpip.so
7ffff71d4000-7ffff73d3000 ---p 00008000 00:71 73004126   /PiP/lib/ldpip.so
7ffff73d3000-7ffff73d4000 r--p 00007000 00:71 73004126   /PiP/lib/ldpip.so
7ffff73d4000-7ffff73d5000 rw-p 00008000 00:71 73004126   /PiP/lib/ldpip.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
7ffff7787000-7ffff778b000 rw-p 00000000 00:00 0
7ffff778b000-7ffff77a2000 r-xp 00000000 fe:01 3050215    /lib64/libpthread-2.28.so
7ffff77a2000-7ffff79a1000 ---p 00017000 fe:01 3050215    /lib64/libpthread-2.28.so
7ffff79a1000-7ffff79a2000 r--p 00016000 fe:01 3050215    /lib64/libpthread-2.28.so
7ffff79a2000-7ffff79a3000 rw-p 00017000 fe:01 3050215    /lib64/libpthread-2.28.so
7ffff79a3000-7ffff79a7000 rw-p 00000000 00:00 0
7ffff79a7000-7ffff79a9000 r-xp 00000000 fe:01 3049753    /lib64/libdl-2.28.so
7ffff79a9000-7ffff7ba9000 ---p 00002000 fe:01 3049753    /lib64/libdl-2.28.so
7ffff7ba9000-7ffff7baa000 r--p 00002000 fe:01 3049753    /lib64/libdl-2.28.so
7ffff7baa000-7ffff7bab000 rw-p 00003000 fe:01 3049753    /lib64/libdl-2.28.so
7ffff7bab000-7ffff7bc5000 r-xp 00000000 00:71 73004125   /PiP/lib/libpip.so.0
7ffff7bc5000-7ffff7dc5000 ---p 0001a000 00:71 73004125   /PiP/lib/libpip.so.0
7ffff7dc5000-7ffff7dc6000 r--p 0001a000 00:71 73004125   /PiP/lib/libpip.so.0
7ffff7dc6000-7ffff7dc7000 rw-p 0001b000 00:71 73004125   /PiP/lib/libpip.so.0
7ffff7dc7000-7ffff7dea000 r-xp 00000000 fe:01 3049541    /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 3049541    /lib64/ld-2.28.so
7ffff7feb000-7ffff7fff000 rw-p 00024000 fe:01 3049541    /lib64/ld-2.28.so
7ffffffde000-7ffffffff000 rw-p 00000000 00:00 0          [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0  [vsyscall]
```

Listing 2.4 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 2.4.

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. Apparently, this API is not thread-safe at all, and thus, the 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.

2.2.4 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[4].

Table 2.2: Compatibility of Tools

Compatible	Incompatible
`strace`	`valgrind`
`ltrace`	

2.3 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.

2.3.1 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

[4] `ltrace` depends on version

of CLONE_FS, CLONE_FILES, CLONE_SIGHAND and CLONE_THREAD are reset, CLONE_VM and CLONE_SYSVSEM is set.

Table 2.3: Differences between two modes

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

Table 2.3 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.

Table 2.4: Mode-Agnostic Functions

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

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

Table 2.5: Execution Mode Predicates

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

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.

2.3.2 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** and **PIP_MODE_PTHREAD**, oring the both, or 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 "**PIP_MODE_PROCESS**" or "**PIP_MODE_PTHREAD**." When the `opts` value is zero or the value of oring 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.

2.4 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()**.

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.

2.4.1 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 **PIP_CPUCORE_ABS**[5] Or, it can be **PIP_CPUCORE_ASIS** to bind to the same CPU cores as the root process calls the spawn function.

2.4.2 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

[5]This is because the core numbers are not contiguous on some CPU architecture, i.e., Fujitsu A64FX CPU.

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 2.5: Before and After Hooks

```
#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[64];
```

```
    sprintf( msg, "Hello from PIPID:%d\n", pipid );
    int len = strlen( msg );
    write( FD_TASK, msg, len );
  }
  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 2.3). Note that the execution of those hook functions are called by the root.

Listing 2.6: Before and After Hooks - Execution

```
$ ./hook 2
PID:3968 MAIN
PID:3969 Before Hook: fd=10
Hello from PIPID:0
PID:3969 After Hook:  fd=10
PID:3970 Before Hook: fd=10
Hello from PIPID:1
PID:3970 After Hook:  fd=10
$
```

2.4.3 PRELOAD

`LD_PRELOAD` does not work when spawning PiP tasks. This is because `dlmopen()` does not support this. Although it is impossible to support the function of the `LD_PRELOAD` without modifying Glibc, PiP library supports similar function, but not exactly the same one with `LD_PRELOAD`.

By setting **PIP_PRELOAD** in the same way of `LD_PRELOAD` (colon (:) separated list of shared objects), those specified object files are loaded before loading user program. At the time of loading objects in the **PIP_PRELOAD**, the Glibc is already loaded, and users cannot override the functions provided by Glibc.

2.5 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 2.7 and 2.8).

Listing 2.7: Function Call of Another

```
#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 goes off the side road, **pip_named_export()** and **pip_named_import()** are called differently than before. The final argument of these functions is actually a format string, just like `printf()`, followed by argument(s) needed by the format.

Listing 2.8: Function Call of Another - Execution

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

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 difficult.

Further, the association of static variables and function addresses heavily depends on the CPU architecture and tool chain. The above description is true on `X86_64` and `AArch64`, however, not true on `X86_32`. Thus, it is not recommended do this.

Rationale

Some readers may wonder why this happens. Let me explain this. This trick is hidden in the address map. Listing 2.5 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). 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 execution code are associated in PIE[6] and PIEprograms and shared objects compiled with the PIEor PICoption can be loaded at any locations.

Unfortunately, this addressing mode is not supported by all CPU architectures[7]. 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 2.7 exhibits differently.

2.6 Debugging Support

Some environment variable settings may help debugging PiP programs.

2.6.1 `PIP_STOP_ON_START`

This environment variable is to stop (by sending `SIGSTOP` to spawned PiP task just before calling the **before hook** (if any). The value of this environment must meet the following format;

```
PIP_STOP_START=[<commandfile>]@<PIPID>
```

The optional ⟨`commandfile`⟩ is a command file or shell script to be executed on the suspension, and ⟨`N`⟩ is the PIPID to be suspended. If ⟨`N`⟩ is -1, then all spawned PiP tasks will be stopped. The ⟨`commandfile`⟩ is invoked with parameters; PID and PIPID of the PiP task, and a path to the program of the task.

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

[7] Listing 2.8 is obtained by running the program on an `X86_64` CPU

Listing 2.9: 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 2.9 shows an example of the script for the **PIP_STOP_ON_START**. Here, **pips** command is invoked instead of some debugging command[8]. Note that the target task is already stopped by the `SIGSTOP` signal. Somehow you have to explicitly deliver the `SIGCONT` signal to the task if you want to resume the task. Listing 2.10 shows the result of **PIP_STOP_ON_START** execution with this script file.

Listing 2.10: 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[4885(R):R] PiP task[2] (PID=4888) is SIGSTOPed and executing 'onstart.cmd' script
Hello World
### onstart.cmd ### hello 4888 2
PID  TID  TT       STAT TIME     PIP COMMAND
4885 4885 pts/0    S+   00:00:00 RL  pip-exec
4888 4888 pts/0    T+   00:00:00 2L  hello
Hello World
$
```

2.6.2 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 PiP-gdb invocation. The value of this environment is as follows;

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

The possible ⟨`SIGNAME`⟩ vale are listed below;

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`.

[8] 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.

Table 2.6: Possible Signal Names for **PIP_GDB_SIGNALS**

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

2.6.3 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 2.4, for example) will be shown.

2.6.4 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.

2.6.5 PIP_GDB_PATH and PIP_GDB_COMMAND (process mode only)

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.

2.7 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*.

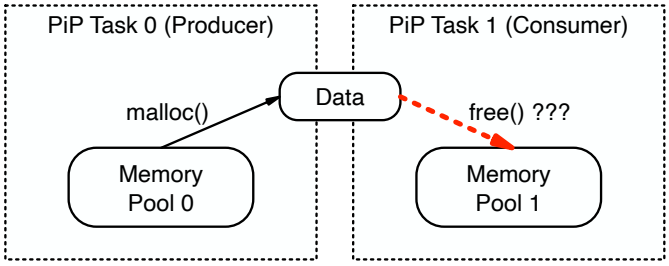

Figure 2.1: Cross-Malloc-Free

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`, but I believe this situation must be avoided.

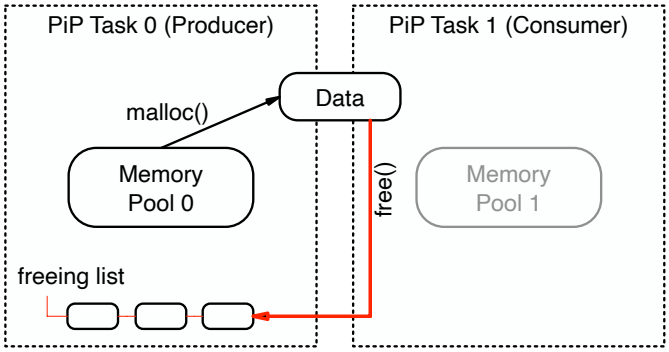

Figure 2.2: Cross-Malloc-Free

To deal with this, PiP library wraps `malloc` routines as shown in Table 2.1. When an memory region is allocated, the `malloc` wrapper function embeds information who allocate the region. When this region is to be `free()`ed, the `free()` wrapper function connects the region to the freeing list of the task allocating the region. The regions in the freeing list are eventually `free()`ed when one of the `malloc` functions is called by the producer (Figure 2.2).

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()` to spawn the PiP task.
4. The before hook is called.
5. Jump into the starting function.

From PiP version 2.4, the wrapper functions listed in Table 2.1 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 inside of **libpip.so**.
4. Load shared objects specified by the **PIP_PRELOAD** environment variable.
5. Load **libpip.so**.
6. Load a user program.
7. Call `clone()` or `pthread_create()` to spawn the PiP task.
8. Jump into a function inside of PiP library and initialize the PiP library.
9. Jump into the start function.

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**.

The Glibc initialization [1] must be done with the execution context 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) jump into 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.

3.2 Remaining Issues

3.2.1 Recycling PiP Tasks

[1] Calling `__ctype_init()`

Index