On Demand Secure Isolation

D1.2 – Isolation API
(public)

Project acronym	ODSI
Project title	On Demand Secure Isolation
Advantage	
Deliverable number	D1.2
Deliverable name	Isolation API
Version	V 1.0
Work package	WP 1 – Minimal Isolation
Lead beneficiary	LILLE
Authors	J. Cartigny, M. Yaker, Q. Bergougnoux, D. Nowak, G. Grimaud (Lille)
Nature	R – Report
Dissemination level	PU – Public
Delivery date	3/1/2018 (M20)

History

Version	Date	Description
0.1	16 March 2017	Preliminary draft
0.9	22 December 2017	Pre-release version
1.0	3 January 2018	Release version

Executive Summary

1 Introduction

PIP is a protokernel: it allows for kernels, ranging from hypervisors to monolithic kernels, to be developed as user mode applications. This means that only PIP is executed in kernel mode (i.e., the privileged mode of the hardware). Indeed, code running in kernel mode has direct access to the whole memory and hardware. It is thus clearly better, from a security point of view, to keep this code as minimal as possible. This stems from the general principle that the trusted computing base (TCB) should be kept minimal. One of the PIP key design is to have a minimal number of functionalities while ensuring strong isolation. More precisely, PIP only manages memory isolation and redirection of interruptions to user space code, and has only 10 system calls. Contrary to micro-kernel, it means that components like scheduler, IPC and authorization are not included in the PIP kernel, neither other mechanism available in monolithic kernel like device abstraction layer, file systems, network stack, etc.

PIP proposes a hierarchic memory isolation model to partition the available memory among several memory-limited space called partition. A partition is a set of physical memory pages mapped to virtual address. At start, all the available memory (as defined inside PIP configuration) makes up the ROOT partition. The root partition can include mapped registers from hardware devices, thus providing access to hardware functionalities. Code executed in a parent partition can request (via system calls to PIP) segregation to its memory space: a new child partition is created on which the parent partition can mapped a subset of its own mapped physical memory pages, thus *delegating* part of its memory space to the child partition. This model is recursive: any partition can create a child partition and delegate part of its memory space. While a partition can read and write in the memory of its child partition (we call this property vertical sharing), sibling partitions cannot access each other's memory (we call this property horizontal isolation).

To manage interruptions, PIP has two different behaviours depending of the kind of interruption:

- Software interrupts (fault or system calls) are relayed to the parent partition (with the exception of the PIP system calls).
- Hardware interrupts are relayed to the ROOT partition which is in charge to forward it to, for instance, a network card driver.

The hierarchic isolation model approach offers a hierarchic **security** model: a child parent partition can only access physical memory pages delegated by its parent. Fur-

thermore, a parent receives all software interruptions from its children[1], thus controlling which interactions have its children with the system.

2 Platforms

Pip, as being a multi-platform kernel, can be run and ported on several architectures. That being said, we currently target two 32 bits platforms:

- x86: Pentium-compatible architecture for desktop computers (Intel Pentium, Core...)
- Intel Galileo Gen 2: x86 Pentium-compatible embedded board (Intel Quark SoC x1000, Pentium-compatible embedded CPU)

Those two platforms, while being similar, have some differences especially when it comes to hardware configuration. For instance, the serial link is configured through IO ports on x86 while being configured through MMIO on the Galileo board.

Both of them still share the same kernel structure and roughly the same hardware abstraction layer and boot sequence, as they share the same CPU architecture.

The x86 architecture version of Pip can also be run on several emulators, including:

- QEMU version 2.7.0
- VirtualBox version 5.1.22
- Bochs version 2.6.9

3 PIP kernel

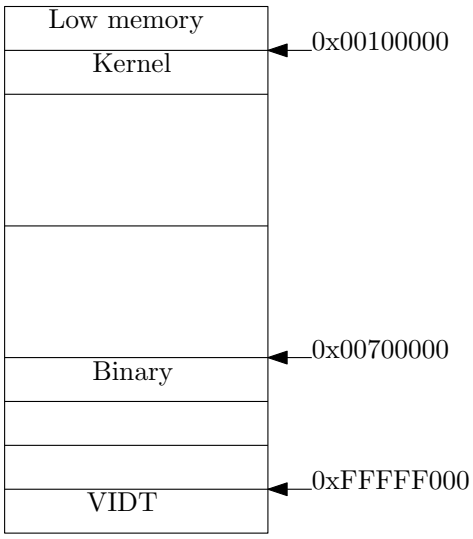

Figure 1: Default mapping of a partition

[1]With the notable exception of the PIP system calls, as a partition can create sub-partitions only based on a (sub)set of its own memory space, thus not breaking the hierarchic isolation model.

3.1 PIP system calls

Pip only provides system calls for the management of partitions of the memory and for the basic dealing of control flow, thus reducing the trusted computing base to its bare minimum. More precisely, Pip provides 10 services that can be called from any partition. Only two of them are for basic control flow, the remaining ones are for memory management.

The services provided by Pip in order to manage partition are described as following:

- createPartition: creates a new child (sub-partition) into the caller (the current partition).
- deletePartition: removes a child partition and puts all its used pages back in the caller.
- addVAddr: maps a page into the given child at a given virtual address.
- pageCount: returns the amount of pages required to configure the internal partition structure in order to map a new page.
- prepare: given the amount of pages returned by the pageCount service, this service prepares a child partition to receive a page map, inserting the corresponding indirection tables.
- collect: checks whether the indirection tables used to translate the given virtual address are empty, and, if so, returns them to the caller.
- removeVAddr: removes a page from a child partition, returning it to the caller.
- mappedInChild: returns the virtual address of the sub-partition in which the given virtual address is mapped.

Pip provides two services related to control flow management, which are the following:

- dispatch: interrupts the caller, and then switches the control flow to the target partition, at the given signal handler.
- resume: resumes the execution of a previously interrupted partition.

A full description of the PIP system calls API is available in appendix A.

3.2 LibPip

LibPip is an appendix project of Pip protokernel. This library, as being a side-project, is not a part of Pip's TCB, and therefore is not concerned by the proof effort on Pip.

3.2.1 LibPip abstraction

The first purpose of LibPip is to offer a set of abstraction function. These functionalities are regrouped into three groups:

- Serial debug functions
- Virtual interrupt management
- Memory and paging management

Serial debug functions Into the LibPip, a minimal set of debug function are implemented. Theses functions send characters over the serial output and they are architecture-dependent.

Virtual interrupt management If a partition has to deal with interruption, you can use this set of function that makes interrupt management easier. These functions are:

- Pip_RegisterInterrupt: register into the VIDT an interrupt handler for a specific interrupt number.
- Pip_VCLI: disable all interrupts.
- Pip_VSTI: enable all interrupts.

In addition to these functions, you can find two C Macros that help you create an interrupt handler. Those macros are not mandatory to implement a proper interrupt management, as you can implement manually your interrupt handler in your desired fashion. In user's C source code:

```
1 /* Initialises an interrupt handler like this */
2 INTERRUPT_HANDLER(assemblyFunctionName, CFunctionName)
3 /* DO SOMETHING */
4
5 END_OF_INTERRUPT
```

And in your assembly code:

```
1 #define __ASSEMBLY__
2 /* As in C source code, initialize like this the assembly part of your interrupt Handler*/
3 INTERRUPT_HANDLER(assemblyFunctionName,CFunctionName)
```

Memory and paging management Pip API could be awkward to use, so we have implemented a set of functions to make memory usage easier.

- Pip_InitPaging: Initialize and organize the available pages.
- Pip_AllocPage: Gives a page from the available pages.
- Pip_FreePage: Frees a page and makes it available again.
- Pip_MapPageWrapper: Map a page into a sub-partition.

Allocating pages Although this mechanism is not provided by Pip itself, and should be implemented in userland instead, LibPip provides a rudimentary page allocator in order to quickly bootstrap any Pip-based project.

A page allocator's goal is to provide an easy way to allocate and free pages in the partition's memory, in order to easily give pages to Pip's API when required. We will be using LibPip's page allocator in the next sections, which provides this API:

```
/* Initializes a page allocator in the given memory region */
void Pip_InitPaging(uint32_t begin, uint32_t end)

/* Allocates a page, returning its address */
uint32_t Pip_AllocPage()

/* Frees a page, returning it back into the allocator pool */
void Pip_FreePage(uint32_t page)
```

While being sufficient to implement small projects, this page allocator is very simple and has several drawbacks:

- No support for multi-level allocators (i.e., bootstrapping the Page Allocator in a child partition while having it already bootstrapped in the parent partition could have undesired side-effects due to memory region overlap).
- No "coherent multi-pages" allocation.
- No support for multiple memory pools.

Initializing the page allocator should be as simple as to invoke `Pip_InitPaging` with the desired memory region. A common invokation is done passing the memory region given by the `pip_fpinfo` structure as parameters, thus initializing the page allocator as one big pool covering the entire free memory area.

Invoking `Pip_AllocPage` should remove a page from the pool and return its address. `Pip_FreePage` takes a page address as a parameter, and puts it back into the allocator's pool.

You can **(and should)** implement your own page allocator in any partition, or port any existing page allocator into a Pip partition instead.

3.2.2 Pip calls

Pip provides 17 functions accessible via hardware specific calls, in the case of x86 processor, these calls are named "far calls". "far call" to a procedure is a call to a function at a different privilege level (For Pip, from User-land to Kernel-land).

```
/* API Calls */
#define Pip_CreatePartition
#define Pip_PageMapCount
#define Pip_Prepare
#define Pip_AddVAddr
#define Pip_Dispatch
#define Pip_Resume
#define Pip_RemoveVAddr
#define Pip_MappedInChild
#define Pip_DeletePartition
#define Pip_Collect

/* x86 hardware access function*/
#define Pip_Outb
```

```
#define Pip_Inb
#define Pip_Outw
#define Pip_Inw
#define Pip_Outl
#define Pip_Inl
#define Pip_OutAddrl
```

The details of all these functions and how to use them are available in the next sections of this document.

3.3 Calling the API without LibPip

You can call Pip services from assembly code by directly invoking far calls as following, for the Pip_CreatePartition call:

```
push    %eax
push    %ebx
push    %ecx
push    %edx
push    %esi
lcall   $0x60, $0
ret
```

(assuming the arguments given to the Pip_CreatePartition parameters are given in the general registers `EAX`, `EBX`, `ECX`, `EDX` and `ESI`.)

4 PIP execution model

4.1 Kernel boot

On x86 architectures, Pip implements the Multiboot specification, which allows any Multiboot-compliant bootloader to boot it on real hardware. The main bootloader we use to boot Pip is GRUB2. A standard entry in `grub.cfg` to boot Pip would be the following:

```
menuentry Pip {
    multiboot /boot/meso.bin
    boot
}
```

Once the bootloader has bootstrapped the kernel, Pip does several initialization operations before starting up the root partition:

- First, it initializes the CPU with required structures (namely, Global Descriptor Table, Interrupt Descriptor Table).
- It then initializes the CPU's optional features, if possible (Global pages, Process context identifiers...).
- It finally sets up the virtual memory environment for the root partition, as well as Pip's required structures.
  - The root partition's memory environment is set as a flat, linear memory space fixing up any memory "hole" present in the physical memory.

– Pip also relocates on boot some hardware which would be inaccessible from the root partition otherwise, such as the VGA terminal controller.

Once everything has been set-up correctly, Pip allocates the stacks for the root partition (kernel stack and user stack), sets up the root partition's Virtual Interrupt Descriptor Table accordingly, and boots the first partition.

4.2 First partition boot

Once the first partition has been started, Pip puts into the EBX general register the address of a structure we call "First Partition Info". This structures contains the following information:

```
#define FPINFO_MAGIC    0xDEADCAFE

typedef struct fpinfo {
    uint32_t magic;         /* Magic number, should be FPINFO_MAGIC */
    uint32_t membegin;      /* Beginning of free memory area */
    uint32_t memend;        /* End of free memory area */
    char revision[64];      /* Pip's build git revision */
} pip_fpinfo;
```

This structure gives an important information: the amount of memory available (from `membegin` to `memend`), for example to set-up a page allocator through LibPip.

To access this structure easily from C code, you would boot the partition by first doing this in assembly code:

```
boot:
    /* Push EBX onto the stack */
    push %ebx

    /* Call main */
    call main
```

And then doing this in C code:

```
#include <pip/fpinfo.h>
#include <pip/paging.h>

void main(pip_fpinfo* fpinfo)
{
    /* First check correctness of FPInfo */
    if(fpinfo->magic == FPINFO_MAGIC) {
        /* Initialize a page allocator from FPInfo */
        Pip_InitPaging(fpinfo->membegin, fpinfo->memend);
    } else {
        DO_SOME_ERROR_HANDLING_STUFF
    }
}
```

4.3 Create a partition

The createPartition service requires 5 virtual addresses as parameters. So, the caller should provide 5 virtual addresses using the pip lib function Pip_allocPage() that gives a single virtual address from the parent partition. Each given virtual address should satisfy the following condition, otherwise, the createPartition will fail.

- Present into the caller

- Accessible into the caller
- Not already given to any other child (sub-partition)
- All virtual addresses should be distinct (do not correspond to the same mapped page into the caller)
- Not a default virtual address

You can now create a sub-partition by executing the following code, which allocates the required pages using LibPip's page allocator (i.e. using pages in the partition's address space, expressed as virtual addresses):

```
#include <pip/api.h>
#include <pip/paging.h>

uint32_t partition; /* We'll store the partition descriptor here */

void makeSubpartition() {
    uint32_t newPart = Pip_AllocPage();     /* Partition Descriptor */
    uint32_t newPd = Pip_AllocPage();       /* Page Directory */
    uint32_t newSh1 = Pip_AllocPage();      /* Shadow 1 */
    uint32_t newSh2 = Pip_AllocPage();      /* Shadow 2 */
    uint32_t newList = Pip_AllocPage();     /* Shadow 3 list */

    /* If createPartition returns 0, we failed */
    if(Pip_CreatePartition(newPart, newPd, newSh1, newSh2, newList))
    {
        partition = newPart;
    } else {
        SOME ERROR OCCURED
    }
}
```

4.4 Map memory pages to partition

The addVaddr (see Appendix C.1). service requires 3 virtual addresses as parameters. Each given virtual address should satisfy a set of conditions, otherwise, this function will fail.

- *source*: The virtual address into the caller to be shared with the sub-partition
  - Present into the caller
  - Accessible into the caller
  - Not already given to any other child (sub-partition) of the caller
  - Not a default virtual address
- *sub-partition*: The virtual address of a sub-partition (its corresponding partition descriptor)
  - Not a default virtual address
- *dest*: The virtual address that will receive the new mapping
  - No existing mapping into this virtual address
  - Not a default virtual address

4.4.1 Step 1: Count

Before the execution of addVaddr we need to check if *sub-partition* is well configured to receive this new mapping. To do that, we execute the Pip service count (see Appendix B.1). This function takes as parameters *sub-partition* and *dest*. It returns 0 if sub-partition is ready to map the given virtual address. In this case we execute immediately the addVaddr service. Otherwise, count returns the amount of pages required to set up the sub-partition. In this case we proceed to step 2 before the execution of addVaddr.

4.4.2 Step 2: Prepare

As explained above, the prepare (see Appendix B.2) Pip service will configure sub-partition in order to receive the new mapping. This service requires 4 parameters. The first two parameters correspond respectively to *sub-partition* and *destination* and should satisfy the same properties already defined. Remaining parameters are detailed below.

- *listVa* : The third parameter is a virtual address such that the corresponding physical address corresponds to the head of a linked list of virtual addresses. The corresponding physical page of each virtual address of this list will be used, as a configuration table, to set up the internal structure of *sub-partition*. Each given virtual address should satisfy the following conditions, otherwise, this function will fail :
  - Present into the caller
  - Accessible into the caller
  - Not already given to any other child (sub-partition)
  - All virtual addresses should be distinct (do not correspond to the same mapped page into the caller)
  - Not the default virtual address (null)

  ⚠ If the user do not provide the right amount of virtual addresses (available through the linked list), this function fails and the configuration of *sub-partition* keeps unchanged or will be partially configured. This is also available if one of the given virtual address do not satisfy one of the listed conditions.

- *isMult* : The last parameter of prepare is a boolean. It should be set to false if the amount of pages returned by count is a multiple of 3 and true otherwise.

⚠ The prepare service could fail if the sub-partition is already configured to receive the corresponding mapping.

In order to map a single virtual address we could execute the following function.

```
1
2 int32_t mapSomething(uint32_t source,
3             uint32_t sub-partition,
4             uint32_t destination)
5 {
6     uint32_t i, count, *page, *tmp;
7
8
9     /*Count call*/
```

```
    if((count = Pip_PageCount((uint32_t)sub-partition,
            (uint32_t)destination)) > 0)
    {
        /* count is the amount of page required by
         Pip to map something into the sub-partition */
        page = Pip_AllocPage();
        *(void**)page = (void*)0;

        /* you have to allocate the required pages */
        for(i=0; i<count-1; i++){
            tmp = Pip_AllocPage();
            *(void**)tmp = page;
            page = tmp;
        }

        /* Prepare the sub-partition to receives the
        pages */
        if (!Pip_Prepare((uint32_t)sub-partition,
            (uint32_t)(destination),
            (uint32_t)page, 0x0))
        {
            ERROR ! SOMETHING WENT WRONG !
        }
    }
    /* map the source data into the sub-partition */
    if(!Pip_AddVaddr((uint32_t)source,
                (uint32_t)sub-partition,
                (uint32_t)destination, 0x1,
                0x1, 0x1)){
        ERROR ! SOMETHING WENT WRONG !
    }

}
```

4.4.3 Map data

After the creation of an "empty" partition (*sub-partition*), we have to link several data like its binary, IDT, etc. To do that, we should use our primitives already detailed above.

Map binary We start by mapping the binary by executing the following code.

```
/* Page size */
#define PAGE_SIZE   0x1000

void mapBinary(
    uint32_t binaryBegin,
    uint32_t binaryLength,
    uint32_t targetPartition,
    uint32_t targetLoadAddress)
{
    /* Map each page of the binary */
    for(uint32_t pos = 0; pos < binaryLength; pos += PAGE_SIZE)
        mapSomething(binaryBegin + pos, targetPartition, targetLoadAddress + pos)
}
```

Map IDT After you have mapped the sub-partition binary, you have to map a virtual IDT for the sub-partition.

```
/* Allocate an empty page for the VIDT */
uint32_t targetVidt = (uint32_t)Pip_AllocPage();
```

It's the same procedure as mapping a binary, you have to create a VIDT structure: VIDT:

- eip: partitionDescriptor
- esp: a stack for the sub-partition execution context
- interrupt flag: set to 0

and map it at 0xFFFFF000 into the sub-partition.

```
#define TARGET_STACK_ADDR   0x2000000
#define PAGE_SIZE           0x1000

*(uint32_t*)targetVidt = targetLoadAddr;

/* Allocate stack */
uint32_t partitionStack = (uint32_t)Pip_AllocPage();

/* Register stack in second entry of the first interrupt descriptor of the VIDT */
*(uint32_t*)(targetVidt + sizeof(uint32_t)) =
        TARGET_STACK_ADDR + PAGE_SIZE - sizeof(uint32_t);

/* Map stack */
mapSomething(partitionStack, targetPartition, TARGET_STACK_ADDR);
```

You can now start the partition by calling the Pip_Dispatch Pip call.

```
/* We use Pip_Notify, which wraps Pip_Dispatch */
/* We send the VINT 0 to the partition, which was set-up before */
Pip_Notify(targetPartition, 0x0, 0x0, 0x0);
```

4.5 Unmap physical memory pages

The removeVaddr service requires 2 virtual addresses as parameters. Each given virtual address should satisfy a set of conditions, otherwise, the removeVaddr service will fail.

- *sub-partition*: The virtual address of a sub-partition
  - Not a default virtual address
- *source*: The virtual address to remove
  - Present into the sub-partition
  - Accessible into the sub-partition
  - Not already given to any other sub-sub-partition by the sub-partition itself.
  - Not a default virtual address

```
#include <pip/api.h>

uint32_t partition; /* We'll store the partition descriptor here */

/* Unmaps a page at address va in the sub-partition */
void unmap(uint32_t va) {
    uint32_t ret_va;
    if(ret_va = Pip_RemoveVAddr(partition, va))
    {
        /* Success, ret_va contains the returned page address in the current partition */
    } else {
        SOME ERROR OCCURED
    }
}
```

4.6 Collect configuration pages

In order to get back some *useless pages*, already given to the kernel in order to set up the internal configuration of a sub-partition (prepare) but no longer useful, we could use the collect service. This function requires 2 virtual addresses as parameters.

- *sub-partition*: The virtual address of a sub-partition.
- *source*: The virtual address into the sub-partition whose corresponding tables will be collected.
  - Not present into the sub-partition
  - All corresponding configuration tables are "empty" (all corresponding virtual addresses are not mapped)
  - Not the default virtual address

```
1  #include <pip/api.h>
2
3  uint32_t partition; /* We'll store the partition descriptor here */
4
5  /* Collects the unused pages */
6  void collect(uint32_t va) {
7      uint32_t ret_va;
8      if(ret_va = Pip_Collect(partition, va))
9      {
10         /* Success, ret_va contains a linked list of returned pages */
11     } else {
12         SOME ERROR OCCURED
13     }
14 }
```

4.7 Delete a partition

To delete a sub-partition you have to call the deletePartition service. This function will fail if the given virtual address do not correspond to a sub-partition (its corresponding partition descriptor).

```
1  #include <pip/api.h>
2
3  uint32_t partition; /* We'll store the partition descriptor here */
4
5  void deleteSubpartition() {
6      uint32_t list;
7      /* If deletePartition returns 0, we failed */
8      if(list = Pip_DeletePartition(newPart))
9      {
10         /* Success: list now contains a linked list of free pages */
11     } else {
12         SOME ERROR OCCURED
13     }
14 }
```

4.8 Register interruptions to callback functions

LibPip provides a Pip_RegisterInterrupt method, which provides an easy way to associate a virtual interrupt to a callback/stack couple.

```
void Pip_RegisterInterrupt(uint32_t int_no, void* callback, uint32_t* stack);
```

This method wraps all the writes to the current partition's Virtual Interrupt Descriptor Table accordingly. For example, if you want to associate the function "handleTimer" to the timer interrupt (virtual interrupt 1), you can do it as following:

```
#define PAGE_SIZE   0x1000              /* The size of a page */

/* data1 contains the interrupted partition address */
void handleTimer(uint32_t data1, uint32_t data2) {
    Pip_Debug_Puts("Timer interrupt handled\n");
}

void registerPartitionInterrupts() {
    /* Allocate an empty page for the interrupt stack */
    uint32_t* interruptStack = Pip_AllocPage();
    if(!interruptStack)
    {
        HANDLE ERROR HERE
    } else {
        Pip_RegisterInterrupt(
            0x1,                                            /* Timer interrupt */
            &handleTimer,                                   /* Timer handler */
            interruptStack + PAGE_SIZE - sizeof(uintptr_t)  /* Interrupt stack */
        );
    }
}
```

You may have noticed the stack initialized. This is because on common architectures, the stack is growing backwards, so when you allocate a new page, you have to set the stack pointer to the end of the page (i.e. the first writable word at the end of the page), not at the beginning, otherwise faults would occur due to stack overflows.

4.9 Switch execution to another partition

Given several partitions are running, you can switch execution to other partition using signals/virtual interrupts. For example, to dispatch a signal to a child partition, and passing it two arguments, you can do as following:

```
uint32_t partition;     /* We suppose there is a valid partition descriptor here */
/* Note: if partition is zero, then Notify will given the signal to the parent */

void notifyPart(uint32_t vint, uint32_t param1, uint32_t param2)
{
    Pip_Notify(
        partition,
        vint,
        param1,
        param2);
}
```

Once this has been done, the execution will switch to the target partition, at the callback and stack previously configured.

You can as well resume the execution of a previously interrupted partition, for example interrupted by a previous Notify call, by doing the following:

```
uint32_t partition;     /* We suppose there is a valid partition descriptor here */
/* Note: if partition is zero, then Resume will try to resume the parent partition */

void resumePart(uint32_t vint_enabled)
{
```

```
    /* PipFlags is 0 when interrupts are enabled, 1 else */
    Pip_Resume(
        partition,
        1 - vint_enabled);
}
```

5 User-space mechanisms

5.1 Child to parent partition IPC

In Pip, a simple notification to parent mechanism was implemented. It allows to a sub-partition to send a message to its parent. This mechanism works through the Pip Interrupt Management System.
Pip_Notify is a simple function, available in Pip Library (Libpip), thereby simplifying notification mechanism between child and parent.

```
uint32_t Pip_Notify(uint32_t destination, uint32_t int_no, uint32_t data1, uint32_t data2);
```

In the parent partition, you have to register an interrupt handler for an arbitrary interrupt value (ex: 66) as explained above.
In the child partition, you have to only call Pip_Notify:

```
Pip_Notify(0, 66, SomeDATA, AnotherData);
```

For a sub-partition, the partition 0 represent its parent. You can send to the parent partition two 32-bits values. For example, you can send a memory range dscription (base address and limit/size) to the parent partition, in order to indicate a shared memory buffer between the parent and the child.

⚠The root partition has no parent, so notifying its parent does not make any sense.

5.2 Parent to children partition IPC

In order to send a message to a partition, you can use, as above, Pip_Notify.
However, the destination parameter is the sub-partition entry value. And in the child you have to register an interrupt handler for an arbitrary value. For example, if the sub-partition entry address is 0x1300000, and in the child, an interrupt handler was registered for 66:

```
Pip_Notify(0x1300000, 66, SomeDATA, AnotherData);
```

5.3 Cross-partition IPC

The Isolation mechanism allow only child-parent and parent-child direct communication. For others kind of communication, you have to implement it by using Pip_Notify function.

For example:

You have this architecture:

For a communication between Partition 1 and Partition 2, you have to implement this mechanism:

We decide that 66 is the interrupt value for communication.
In partition 1 and 2, implement an interrupt handler for 66 and register it into the VIDT.

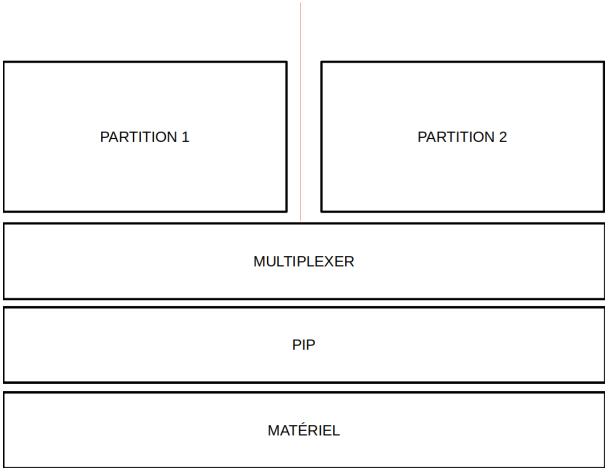

Figure 2:

In the multiplexer, also implement an interrupt handler for 66 and with the purpose to check the communications rights of a partition and dispatch the communication to the other one.
As seen above, you can send two parameters. In partition 1, we can imagine a call to Pip_Notify like this:

```
1 Pip_Notify(0, 66, SomeData, PARTITION2);
```

In the multiplexer 66-handler, we can see:

```
1 Pip_Notify(PARTITION2_ADDRESS, 66, SomeData, SENDER);
```

(SENDER is partition 1).

And partition 2 can reply by sending:

```
1 Pip_Notify(0, 66, ACK_value, SENDER);
```

5.4 Hardware access

If a partition is allowed to access a specific hardware, you have to use Pip's library function to do it. In the case of x86 architecture, the I/0 functions are available through a set of abstractions.

- Pip_Outb, Pip_Outw, Pip_Outl
- Pip_Inb, Pip_Inw, Pip_Inl
- Pip_Outaddrl

For example, the x86 serial link can be easily programmed to output a character on the serial line this way:

```
1 #include <pip/api.h>
2
3 /* Serial link base IO port */
```

```
#define SERIAL_PORT_BASE 0x3f8

void serial_putchar(char c)
{
    /* Check whether the link is ready or not */
    while (!(Pip_Inb(SERIAL_PORT_BASE + 5) & 0x20));

    /* Print the character on the serial line */
    Pip_Outb(SERIAL_PORT_BASE, c);
}
```

5.5 Implementing a simple scheduler

Implementing a scheduler using Pip should rely mainly on Pip's Notify and Resume calls.

Given a hardware timer interrupt should trigger the virtual interrupt 1 in the root partition (Multiplexer), and interrupt the execution of the previously running partition, a very simple scheduler could be implemented as following:

```
/* Let's say we have two FreeRTOS instances running */
uint32_t freeRTOS1, freeRTOS2;

/* Here, 1 is the VINT corresponding to the Timer interrupt. */
void handleTimer(uint32_t caller, uint32_t data)
{
    /* Switch execution according to the caller */
    if(caller == freeRTOS1)
        Pip_Notify(freeRTOS2, 0x1, 0x0, 0x0);
    else
        Pip_Notify(freeRTOS1, 0x1, 0x0, 0x0);
}
```

You can easily implement a round-robin-like scheduler or various scheduling models using the same concepts.

Appendices

A Creating and removing partitions

A.1 Create Partition

Creates a new child partition. Usage:

```
uint32_t ret = createPartition(
    descriptor,
    pageDir,
    sh1,
    sh2,
    sh3
);
```

`ret` contains 0 if the call failed, 1 else.

The five empty pages given as arguments should become the following once the partition has been created:

- `descriptor`: Partition Descriptor for the newly-created partition

- `pageDir`: MMU configuration root page
- `sh1`: Clone of the MMU configuration storing information related to page derivation
- `sh2`: Clone of the MMU configuration storing information related to the parent partition
- `list`: Linked list of physical-virtual addresses couples, storing the address translations of the MMU configuration's pages in the parent partition

A.2 Delete Partition

Removes a child partition, freeing all its pages and giving them back to the caller. Usage:

```
1 uint32_t ret = deletePartition(
2     descriptor
3 );
```

`ret` contains 0 if the call failed, the address on a page containing a linked list of the freed pages else. The linked lists of pages taken or returned by Pip share the same structure. Each page contains a pointer to the next one, while the last one holds a null pointer.

For example, you can build a list of three pages as following:

```
1 uint32_t *pg1, *pg2, *pg3;
2 *pg1 = pg2;
3 *pg2 = pg3;
4 *pg3 = 0;
```

- `descriptor`: Partition Descriptor of the partition to remove

B Managing the partition's internals

B.1 Page Count

Returns the amount of pages required to prepare a child partition before mapping a page. Usage:

```
1 uint32_t ret = pageCount(
2     descriptor,
3     targetAddr
4 );
```

`ret` contains the amount of pages required (0 if the partition is already prepared, the amount of required pages else).

- `descriptor`: Partition Descriptor of the partition we plan to prepare
- `targetAddr`: Target virtual address in the child partition

B.2 Prepare

Prepares a child partition to receive a mapping. Usage:

```
1 uint32_t ret = prepare(
2     descriptor,
3     targetAddr,
4     list
5 );
```

`ret` returns 0 if the call failed, 1 else.

- `descriptor`: Partition Descriptor of the partition we want to prepare
- `targetAddr`: Target virtual address in the child partition
- `list`: A pointer to a page containing a linked list of pages

B.3 Collect

Retrieves empty MMU configuration pages and give them back to the caller. Usage:

```
1 uint32_t ret = collect(
2     descriptor,
3     targetAddr
4 );
```

`ret` returns 0 if the call failed, the address of a linked list of freed pages else.

- `descriptor`: Partition Descriptor of the partition we want to clean
- `targetAddr`: Target virtual address in the child partition

C Managing pages

C.1 Add VAddr

Maps a page into a child partition. Usage:

```
1 uint32_t ret = addVaddr(
2     page,
3     targetPartition,
4     targetAddr
5 );
```

`ret` returns 0 if the call failed, 1 else.

- `page`: The page we want to give to a child
- `targetPartition`: Partition Descriptor of the child partition
- `targetAddr`: The address at which to place the page

C.2 Remove VAddr

Removes a page from a child partition, and gives it back to the caller. Usage:

```
uint32_t ret = removeVaddr(
    targetPartition,
    targetAddr
);
```

`ret` returns 0 if the call failed, the address of the returned page in the caller else.

- `targetPartition`: Partition Descriptor of the child partition
- `targetAddr`: The address of the target page we want to get back

D Managing control flow

D.1 Dispatch

Dispatches a signal to a partition related to the caller (parent or child). Usage:

```
dispatch(
    partition,
    signal
);
```

- `partition`: Partition Descriptor of the target partition (0 for parent)
- `signal`: The virtual interrupt number of the signal (e.g. 1 for timer, 2 for keyboard...)

`Dispatch` saves the interrupted context onto the caller's stack or its VIDT, depending on its current state (virtual interrupts enabled or disabled, stack overflowing...), and then immediately gives the execution to the target partition's signal handler, if the signal is handled, thus being similar to the `INT` instruction.

If, for whatever reason, the signal is not handled, then dispatch does absolutely nothing.

D.2 Resume

Resumes the execution of a previously interrupted partition. Usage:

```
resume(
    partition,
    intstate
);
```

- `partition`: Partition Descriptor of the target partition (0 for parent)
- `intstate`: 1 if we enable virtual interrupts after resuming, 0 else.

`Resume` does **NOT** save the context of the caller, being somehow similar to the `IRET` instruction. It immediatly resumes the interrupted state of the target partition, and gives back execution to the latter.

E Managing hardware

E.1 IO ports

Those functions provide many ways to access the IO ports on the x86 architecture. Usage:

```
outb(
    port,
    value
);

ret = inb(
    port
);
```

Given you're using an `IN` operation, `ret` contains the value stored into the IO port.

The arguments given to the various IO port operations are exactly the same as if you were using them directly in assembly through the `INB`/`OUTB`/`INW`/`OUTW`/`INL`/`OUTL`... operations.

- `port`: The IO port to read/write from/to
- `value`: Given you're using an `OUT` operation, the value to write onto the IO port