

μ PandOS: Phase 1

Luca Bassi (luca.bassi14@studio.unibo.it)
Gabriele Genovese (gabriele.genovese2@studio.unibo.it)

November 6, 2023

Introduction

The μ PandOS operating system described below is inspired to the previous experience of PandOSplus and AMIKaya OS.

1 Phase 1 - Level 2: The Queue Managers

Level 2 of μ PandOS instantiates the key operating system concept that active entities at one layer are just data structures at lower layers. In this case, the active entities at a higher level are processes (i.e. programs in execution) and the data structure(s) that represent them at this level are *Process Control Blocks* (PCBs).

```
1 /* process table entry type */
2 typedef struct pcb_t
3 {
4     /* process queue */
5     struct list_head p_list;
6
7     /* process tree fields */
8     struct pcb_t *p_parent; /* ptr to parent */
9     struct list_head p_child; /* children list */
10    struct list_head p_sib; /* sibling list */
11
12    /* process status information */
13    state_t p_s; /* processor state */
14    cpu_t p_time; /* cpu time used by proc */
15
16    /* First message in the message queue */
17    struct list_head msg_inbox;
18
19    /* Pointer to the support struct */
20    support_t *p_supportStruct;
21
22    /* process id */
23    int p_pid;
24 } pcb_t, *pcb_PTR;
```

The Process Queue Manager will implement three PCB related sets of functions:

- The allocation and deallocation of PCBs.
- The maintenance of queues of PCBs.
- The maintenance of trees of PCBs.

Modules will communicate with the kernel through message passing. At this level, messages are represented by a message block (`msg_t`).

```

1  /* message entry type */
2  typedef struct msg_t
3  {
4      /* message queue */
5      struct list_head m_list;
6
7      /* thread that sent this message */
8      struct pcb_t *m_sender;
9
10     /* the payload of the message */
11     unsigned int m_payload;
12 } msg_t, *msg_PTR;

```

The Message Queue Manager will implement functions to provide these services:

- Allocation/deallocation of single message elements.
- Maintenance of messages' queues.

2 Processes

2.1 The Allocation and Deallocation of PCBs

One may assume that μ PandOS supports no more than `MAXPROC` concurrent processes; where `MAXPROC` should be set to 40 (in the `const.h` file). Thus this level needs a “pool” of `MAXPROC` PCBs to allocate from and deallocate to. Assuming that there is a set of `MAXPROC` PCBs, the free or unused ones can be kept on a double, circularly linked list (using the `p_list` field), called the `pcbFree` list, whose head is pointed to by the variable `pcbFree_h`.

To support the allocation and deallocation of PCBs there should be the following three externally visible functions:

- PCBs which are no longer in use can be returned to the `pcbFree_h` list by using the method:

```
void freePcb(pcb_t *p)
```

Insert the element pointed to by `p` onto the `pcbFree` list.

- PCBs should be allocated by using:

```
pcb_t *allocPcb()
```

Return `NULL` if the `pcbFree` list is empty. Otherwise, remove an element from the `pcbFree` list, provide initial values for ALL of the PCBs fields (i.e. `NULL` and/or 0) and then return a pointer to the removed element. PCBs get reused, so it is important that no previous value persist in a PCB when it gets reallocated.

There is still the question of how one acquires storage for `MAXPROC` PCBs and gets these `MAXPROC` PCBs initially onto the `pcbFree` list. Unfortunately, there is no `malloc()` feature to acquire dynamic (i.e. non-automatic) storage that will persist for the lifetime of the OS and not just the lifetime of the function they are declared in. Instead, the storage for the `MAXPROC` PCBs will be allocated as static storage. A `static pcb_t pcbTable[MAXPROC]` will be declared in `initPcbs()`. Furthermore, this method will insert each of the `MAXPROC` PCBs onto the `pcbFree` list.

- To initialize the `pcbFree` list:

```
initPcbs()
```

Initialize the `pcbFree` list to contain all the elements of the static array of `MAXPROC` PCBs. This method will be called only once during data structure initialization.

2.2 Process Queue Maintenance

The methods below do not manipulate a particular queue or set of queues. Instead they are generic queue manipulation methods; one of the parameters is a pointer to the queue upon which the indicated operation is to be performed.

The queues of PCBs to be manipulated, which are called process queues, are all double, circularly linked lists, via the `p_list` field.

To support process queues there should be the following externally visible functions:

void mkEmptyProcQ(struct list_head *head)

This method is used to initialize a variable to be head pointer to a process queue.

int emptyProcQ(struct list_head *head)

Return TRUE if the queue whose head is pointed to by `head` is empty. Return FALSE otherwise.

void insertProcQ(struct list_head *head, pcb_t *p)

Insert the PCB pointed by `p` into the process queue whose head pointer is pointed to by `head`.

pcb_t *headProcQ(struct list_head *head)

Return a pointer to the first PCB from the process queue whose head is pointed to by `head`. Do not remove this PCB from the process queue. Return NULL if the process queue is empty.

pcb_t *removeProcQ(struct list_head *head)

Remove the first (i.e. head) element from the process queue whose head pointer is pointed to by `head`. Return NULL if the process queue was initially empty; otherwise return the pointer to the removed element.

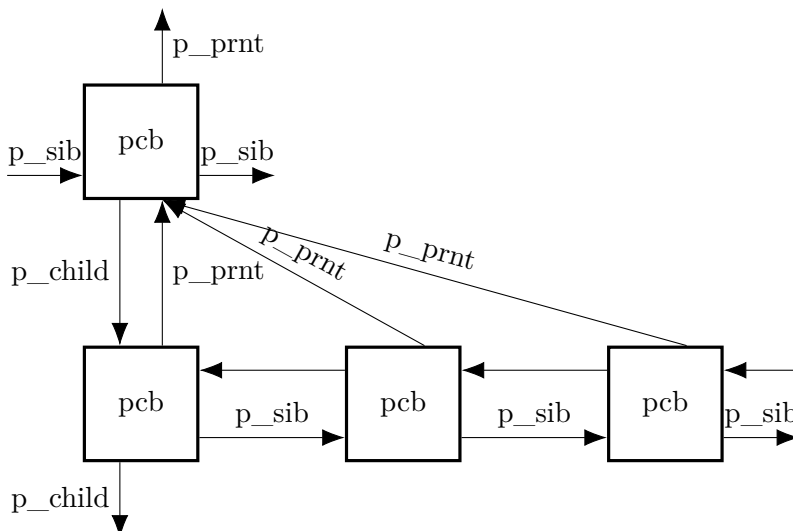
pcb_t *outProcQ(struct list_head *head, pcb_t *p)

Remove the PCB pointed to by `p` from the process queue whose head pointer is pointed to by `head`. If the desired entry is not in the indicated queue (an error condition), return NULL; otherwise, return `p`. Note that `p` can point to any element of the process queue.

2.3 Process Tree Maintenance

In addition to possibly participating in a process queue, PCBs are also organised into trees of PCBs, called *process trees*. `p_prnt`, `p_child`, and `p_sib` are used for this purpose.

The process trees should be implemented as follows. A parent PCB contains a list of children PCBs (`p_child`), each child process `p_sib` can be used to add the child to this list. Each child process has a pointer to its parent PCB (`p_prnt`).



To support process trees there should be the following externally visible functions:

int emptyChild(pcb_t *p)

Return TRUE if the PCB pointed to by **p** has no children. Return FALSE otherwise.

void insertChild(pcb_t *prnt, pcb_t *p)

Make the PCB pointed to by **p** a child of the PCB pointed to by **prnt**.

pcb_t *removeChild(pcb_t *p)

Make the first child of the PCB pointed to by **p** no longer a child of **p**. Return NULL if initially there were no children of **p**. Otherwise, return a pointer to this removed first child PCB.

pcb_t *outChild(pcb_t *p)

Make the PCB pointed to by **p** no longer the child of its parent. If the PCB pointed to by **p** has no parent, return NULL; otherwise, return **p**. Note that the element pointed to by **p** could be in an arbitrary position (i.e. not be the first child of its parent).

3 Messages

3.1 The Allocation and Deallocation of Messages

One may assume that μ PandOS supports no more than **MAXMESSAGES** concurrent messages; where **MAXMESSAGES** should be set to 40 (in the **const.h** file). Thus this level needs a “pool” of **MAXMESSAGES** messages to allocate from and deallocate to. Assuming that there is a set of **MAXMESSAGES** messages, the free or unused ones can be kept on a double, circularly linked list (using the **m_list** field), called the **msgFree** list, whose head is pointed to by the variable **msgFree_h**.

To support the allocation and deallocation of messages there should be the following three externally visible functions:

- Messages which are no longer in use can be returned to the **msgFree_h** list by using the method:

void freeMsg(msg_t *m)

Insert the element pointed to by **m** onto the **msgFree** list.

- Messages should be allocated by using:

msg_t *allocMsg()

Return NULL if the **msgFree** list is empty. Otherwise, remove an element from the **msgFree** list, provide initial values for ALL of the messages fields (i.e. NULL and/or 0) and then return a pointer to the removed element. Messages get reused, so it is important that no previous value persist in a message when it gets reallocated.

There is still the question of how one acquires storage for **MAXMESSAGES** messages and gets these **MAXMESSAGES** messages initially onto the **msgFree** list. Unfortunately, there is no **malloc()** feature to acquire dynamic (i.e. non-automatic) storage that will persist for the lifetime of the OS and not just the lifetime of the function they are declared in. Instead, the storage for the **MAXMESSAGES** messages will be allocated as static storage. A **static msg_t msgTable[MAXMESSAGES]** will be declared in **initMsgs()**. Furthermore, this method will insert each of the **MAXMESSAGES** messages onto the **msgFree** list.

- To initialize the **msgFree** list:

initMsgs()

Initialize the **msgFree** list to contain all the elements of the static array of **MAXMESSAGES** messages. This method will be called only once during data structure initialization.

3.2 Message Queue Maintenance

The functions below do not manipulate a particular message queue or set of queues. Instead they are generic queue manipulation methods; one of the parameters is a pointer to the message queue upon which the indicated operation is to be performed.

Queues to be manipulated are circular, double linked and head pointed lists.

To provide support to message queues, the following externally visible functions should be implemented:

void mkEmptyMessageQ(struct list_head *head)

Used to initialize a variable to be head pointer to a message queue; returns a pointer to the head of an empty message queue, i.e. NULL.

int emptyMessageQ(struct list_head *head)

Returns TRUE if the queue whose tail is pointed to by **head** is empty, FALSE otherwise.

void insertMessage(struct list_head *head, msg_t *m)

Insert the message pointed to by **m** at the end of the queue whose head pointer is pointed to by **head**.

void pushMessage(struct list_head *head, msg_t *m)

Insert the message pointed to by **m** at the head of the queue whose head pointer is pointed to by **head**.

msg_t *popMessage(struct list_head *head, pcb_t *p_ptr)

Remove the first element (starting by the head) from the message queue accessed via **head** whose sender is **p_ptr**.

If **p_ptr** is NULL, return the first message in the queue. Return NULL if the message queue was empty or if no message from **p_ptr** was found; otherwise return the pointer to the removed message.

msg_t *headMessage(struct list_head *head)

Return a pointer to the first message from the queue whose head is pointed to by **head**. Do not remove the message from the queue. Return NULL if the queue is empty.

4 Nuts and Bolts

There is no one right way to implement the functionality of this level. The recommended approach is to create two modules (i.e. files): one for the Message Queue and one for PCB initialisation/allocation/deallocation, process queue maintenance, and process tree maintenance. The second module, **pcb.c**, in addition to the public and HIDDEN/private helper functions, will also contain the declaration for the **pcbTable** and for the private global variable that points to the head of the **pcbFree** list.

```
static pcb_t pcbTable[MAXPROC];
```

```
LIST_HEAD(pcbFree_h);
```

```
static int next_pid = 1;
```

The Message queue module, **msg.c**, in addition to the public and HIDDEN/private helper functions, will also contain the declarations for **msgTable** and **msgFree_h**

```
static msg_t msgTable[MAXMESSAGES];
```

```
LIST_HEAD(msgFree_h);
```

The declarations for **pcb_t** and **msg_t** would then be placed in the **types.h** file. This is because many other modules will need to access these definitions.

5 Testing

There is a provided test file, `p1test.c`, that will check the behaviour of your code.

As with any non-trivial system, you are strongly encouraged to use the `make` program to maintain your code. A sample `Makefile` has been supplied for you to use. See Chapter 10 in the POPS reference for more compilation details.

Once your source files have been correctly compiled, linked together (with appropriate linker script, `crtso.o`, and `libumps.o`), and post-processed with `umps3-elf2umps` (all performed by the sample `Makefile`), your code can be tested by launching the `μMPS3` emulator.

At a terminal prompt, enter: `umps3`. Create a new machine configuration in the project folder.

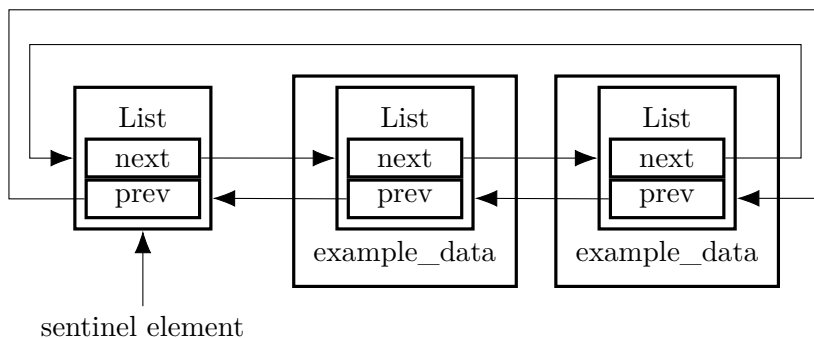
The test program reports on its progress by writing messages to `Windows > Terminal 0`. These messages are also added to one of two memory buffers; `errbuf` for error messages and `okbuf` for all other messages. At the conclusion of the test program, either successful or unsuccessful, `μMPS3` will display a final message and then enter an infinite loop. The final message will either be `System Halted` for successful termination, or `Kernel Panic` for unsuccessful termination (you could have to click the play button a second time if Exceptions is enable in the emulator Stop Mask).

Linux kernel lists

We will use the generic and type-oblivious lists of the Linux kernel. If it's needed to create a list for an existing data type, just add a `list_head` field in the struct (this means that the `list_head` struct is type-oblivious). These are double linked lists, so every element of the list is pointed to the next and previous element in the list. There is a sentinel element that isn't an element of the list, but it's used to link the first and last elements of the list. This data type comes with a useful range of macros and functions, located in the `listx.h` file (it's suggested to take a look at the file and read the code to understand its functionality).

```
struct list_head {
    struct list_head *next;
    struct list_head *prev;
}

struct example_data {
    int value;
    // ...
    struct list_head e_list;
}
```



```
#define LIST_HEAD_INIT(name)
    Macro to create an empty list.
```

```
#define LIST_HEAD(name)
    Macro to create an empty list (declare also the variable).
```

```

void INIT_LIST_HEAD(struct list_head *list)
    Inline function to create an empty list.

int list_empty(const struct list_head *head)
    Return 1 if the list pointed by head is empty.

void list_add(struct list_head *new, struct list_head *head)
    Add element point by new to the head of list pointed by head.

void list_add_tail(struct list_head *new, struct list_head *head)
    Add element point by new to the tail of list pointed by head.

void __list_add(struct list_head *new, struct list_head *prev, struct list_head *next)
    Add element point by new in an arbitrary position of list pointed by head.

void list_del(struct list_head *entry)
    Remove element point by entry from the list in which it is contained.

#define container_of(ptr, type, member)
    Macro to get the struct containing the list_head pointed by ptr, type is the struct name that
    contain the list_head, member is the name of the list_head inside the struct.

#define list_for_each(pos, head)
    Scroll through the list pointed by head, the current position will be pointed by pos.

#define list_for_each_entry(pos, head, member)
    Scroll through the content of the list pointed by head, the current element will be pointed by
    pos, member is the name of the list_head inside the struct.

```

Credits

This document took inspiration from Enrico Cataldi's "Guide to the AMIKaya Operating System Project" and the Principles of Operation book.

μ PandOS: Phase 2

Luca Bassi (luca.bassi14@studio.unibo.it)
Gabriele Genovese (gabriele.genovese2@studio.unibo.it)

March 7, 2024

Phase 2 - Level 3: The Nucleus

Level 3, the Nucleus, builds on the previous levels in two key ways:

1. Receives the control flow from the exception handling facility of Level 1 (the ROM). There are two categories of exceptions [Chapter 3-pops]:
 - TLB-Refill events, a relatively frequent occurrence which is triggered during address translation when no matching entries are found in the TLB. Since address translation will not be introduced until the Support Level, the handling of TLB-Refill events is delayed until then.
 - All other exception types, including device/timer interrupts, which, by definition, occur infrequently. This category can be further broken down into:
 - Interrupts: peripheral devices and internal timers
 - System Service calls (SYSCALL)
 - TLB exceptions - exceptions related to the memory management unit (MMU)
 - Program Trap exceptions (e.g. Bus Error)
2. Using the data structures from Level 2 (Phase 1) and the facility to handle both system service calls and device interrupts, timer interrupts in particular, provide a process scheduler that support multiprogramming.

Hence, the purpose of the Nucleus is to provide an environment in which asynchronous sequential processes (i.e. heavyweight threads) exist, each making forward progress as they take turns sharing the processor. Furthermore, the Nucleus provides these processes with exception handling routines, low-level synchronization primitives, and a facility for “passing up” the handling of Program Trap, TLB exceptions and certain SYSCALL requests to the Support Level (Phase 3).

Important: Since virtual memory is not supported until the Support Level (Phase 3), all addresses at this level are assumed to be physical addresses.

In summary, after some one-time Nucleus initialization code, the Nucleus will repeatedly dispatch a process, i.e. remove a PCB from the Ready Queue and perform a LDST (load state) on the processor state stored in the PCB (`p_s`). This Current Process will run until:

- It makes a system call (SYSCALL). The Nucleus will handle the system call or pass along the handling to the Support Level. Some system calls block the Current Process: the Scheduler is called to dispatch the next job. If the system call is non-blocking, the control flow is returned to the Current Process.
- It terminates. The Nucleus will call the Scheduler to dispatch the next process on the Ready Queue.

- The timer assigned to the Scheduler generates an interrupt; the Current Process's quantum/time slice has expired. Its PCB is enqueued back on the Ready Queue and the Scheduler is called to dispatch the next job.
- A device interrupt occurs (exclusive of the timer assigned to the Scheduler). The interrupt is acknowledged, and the device's status code is passed along to the PCB (i.e. process) that got unblocked as a result of the interrupt; the PCB that was waiting for the I/O to complete. The newly unblocked PCB is enqueued back on the Ready Queue and the control flow is returned to the Current Process.

If the Scheduler ever discovers that the Ready Queue is empty it will either HALT execution (if there are no more processes to run), WAIT for an I/O operation to complete (which will unblock a PCB and populate the Ready Queue), or PANIC (halt execution in the presence of deadlock).

μ PandOS is a microkernel. Hence, there will be only a few system calls handled by the nucleus. A System Service Interface (SSI) will provide other fundamental kernel's services.

Hence, the Nucleus's functionality can be broken down into six main categories:

- Nucleus initialization [Section 1].
- The Scheduler [Section 2].
- Exception handling and SYSCALL processing [Section 5].
- Device interrupt handler [Section 8].
- The passing up of the handling of all other events. This includes TLB-Refill events [Section 3], SYSCALLs not handled at this level, page faults, Program Trap exceptions, etc. [Section 9].
- The SSI handler [Section 6].

Important: In this phase, there will be some differences between μ MPS3 and μ RISCV. In this document reference will be made to an implementation for μ MPS3. For the differences between μ MPS3 and μ RISCV, see section 12.

1 Nucleus Initialization

Every program needs an entry point (i.e. `main()`). The entry point for μ PandOS performs the Nucleus initialization, which includes:

1. Declare the Level 3 global variables. This should include:
 - Process Count: integer indicating the number of started, but not yet terminated processes.
 - Soft-block Count: A process can be either in the "ready", "running", or "blocked" (also known as "waiting") state. This integer is the number of started, but not terminated processes that in are the "blocked" state due to an I/O or timer request.
 - Ready Queue: Tail pointer to a queue of PCBs that are in the "ready" state.
 - Current Process: Pointer to the PCB that is in the "running" state, i.e. the current executing process.
 - Blocked PCBs: The Nucleus maintains one list of blocked PCBs for each external (sub)device (or an array of length `SEMDEVLEN - 1` of `pcb_t` pointers), plus one additional list to support the Pseudo-clock [Section 8.3]. Since terminal devices are actually two independent sub-devices, the Nucleus maintains two lists/pointers for each terminal device [Section 5.7-pops].

2. Populate the Processor 0 Pass Up Vector. The Pass Up Vector is part of the BIOS Data Page, and for Processor 0, is located at 0x0FFF.F900 (constant `PASSUPVECTOR`) [Section 8.5-pops]. The Pass Up Vector is where the BIOS finds the address of the Nucleus functions to pass control to for both TLB-Refill events and all other exceptions. Specifically,

- Set the Nucleus TLB-Refill event handler address to

```
passupvector->tlb_refill_handler = (memaddr)uTLB_RefillHandler;
```

where `memaddr`, in `types.h`, has been aliased to `unsigned int`. Since address translation is not implemented until the Support Level, `uTLB_RefillHandler` is a place holder function whose code is provided [Section 3]. This code will then be replaced when the Support Level is implemented.

- Set the Stack Pointer for the Nucleus TLB-Refill event handler to the top of the Nucleus stack page: 0x2000.1000 (constant `KERNELSTACK`). Stacks in μ MPS3 grow down.
- Set the Nucleus exception handler address to the address of your Level 3 Nucleus function (e.g. `exceptionHandler`) that is to be the entry point for exception (and interrupt) handling [Section 4]:

```
passupvector->exception_handler = (memaddr)exceptionHandler;
```

- Set the Stack pointer for the Nucleus exception handler to the top of the Nucleus stack page: 0x2000.1000 (constant `KERNELSTACK`).

3. Initialize the Level 2 (Phase 1) data structures:

```
initPcbs();  
initMsgs();
```

4. Initialize all the previously declared variables;

5. Load the system-wide Interval Timer with 100 milliseconds (constant `PSECOND`) [Section 8.3].

6. Instantiate a first process, place its PCB in the Ready Queue, and increment Process Count. A process is instantiated by allocating a PCB (i.e. `allocPcb()`), and initializing the processor state that is part of the PCB. In particular this process needs to have interrupts enabled, kernel-mode on, the SP set to `RAMTOP` (i.e. use the last RAM frame for its stack), and its PC set to the address of `SSI_function_entry_point`. Furthermore, set the remaining PCB fields as follows:

- Set all the Process Tree fields to NULL.
- Set the accumulated time field (`p_time`) to zero.
- Set the Support Structure pointer (`p_supportStruct`) to NULL.

Important: When setting up a new processor state one must set the previous bits (i.e. `IEp` & `KUp`) and not the current bits (i.e. `IEc` & `KUc`) in the Status register for the desired assignment to take effect after the initial LDST loads the processor state [Section 7.4-pops]. Test is a supplied function/process that will help you debug your Nucleus. One can assign a variable (i.e. the PC) the address of a function by using

```
ssi_pcb->p_s.s_pc = (memaddr) SSI_function_entry_point;
```

For rather technical reasons, whenever one assigns a value to the PC one must also assign the same value to the general purpose register `t9` (a.k.a. `s_t9` as defined in `types.h`) [Section 10.2-pops].

7. Instantiate a second process, place its PCB in the Ready Queue, and increment Process Count. A process is instantiated by allocating a PCB (i.e. `allocPcb()`), and initializing the processor state that is part of the PCB. In particular this process needs to have interrupts enabled, the processor Local Timer enabled, kernel-mode on, the SP set to `RAMTOP - (2 * FRAMESIZE)` (i.e. use the last RAM frame for its stack minus the space needed by the first process), and its PC set to the address of `test`. Set the remaining PCB fields as for the first process. Remember to declare `test` as “external” in your program by including the line:

```
extern void test();
```

8. Call the Scheduler.

Once `main()` calls the Scheduler its task is complete since control should never return to `main()`. At this point the only mechanism for re-entering the Nucleus is through an exception; which includes device interrupts. As long as there are processes to run, the processor is executing instructions on their behalf and only temporarily enters the Nucleus long enough to handle a device interrupt or exception when they occur. At boot/reset time the Nucleus is loaded into RAM beginning with the second frame of RAM: 0x2000.1000. The first frame of RAM is reserved for the Nucleus stack. Furthermore, Processor 0 will be in kernel-mode with all interrupts masked, and the processor Local Timer disabled. The PC is assigned 0x2000.1000 and the SP, which was initially set to 0x2000.1000 at boot-time, will now be some value less, due to the activation record for `main()` that now sits on the stack [Section 8.2-pops].

2 The Scheduler

Your Nucleus should guarantee finite progress; consequently, every ready process will have an opportunity to execute. The Nucleus should implement a simple preemptive round-robin scheduling algorithm with a time slice value of 5 milliseconds (constant `TIMESLICE`). Preemptive CPU scheduling requires the use of an interrupt generating system clock. μ MPS3 offers two choices: the single system-wide Interval Timer or a Processor’s Local Timer (PLT) [Section 4.1-pops]. One should use the PLT to support per processor scheduling since the Interval Timer is reserved for implementing Pseudo-clock ticks [Section 8.3]. In its simplest form whenever the Scheduler is called it should dispatch the “next” process in the Ready Queue.

1. Remove the PCB from the head of the Ready Queue and store the pointer to the PCB in the Current Process field.
2. Load 5 milliseconds on the PLT [Section 4.1.4-pops].
3. Perform a Load Processor State (LDST) on the processor state stored in PCB of the Current Process (`p_s`).

Dispatching a process transitions it from a “ready” process to a “running” process.

The Scheduler should behave in the following manner if the Ready Queue is empty:

1. If the Process Count is 1 and the SSI is the only process in the system, invoke the `HALT BIOS` service/instruction [Section 7.3.7-pops]. Consider this a job well done!
2. If the Process Count > 1 and the Soft-block Count > 0 enter a Wait State. A Wait State is where the processor is not executing instructions, but “twiddling its thumbs” waiting for a device interrupt to occur. μ MPS3 supports a `WAIT` instruction expressly for this purpose [Section 7.2.2-pops].

Important: Before executing the `WAIT` instruction, the Scheduler must first set the Status register to enable interrupts and either disable the PLT (also through the Status register), or load it with a very large value. The first interrupt that occurs after entering a Wait State should not be for the PLT.

3. Deadlock for μ PandOS is defined as when the Process Count > 0 and the Soft-block Count is 0. Take an appropriate deadlock detected action; invoke the PANIC BIOS service/instruction [Section 7.3.6-pops].

3 TLB-Refill events

As outlined above [Section 1], the Processor 0 Pass Up Vector's Nucleus TLB-Refill event handler address should be set to the address of your TLB-Refill event handler (e.g. `uTLB_RefillHandler`). The code for this function, for Level 3/Phase 2 testing purposes should be as follows:

```
void uTLB_RefillHandler() {
    setENTRYHI(0x80000000);
    setENTRYLO(0x00000000);
    TLBWR();
    LDST((state_t*) 0x0FFF.F000);
}
```

Writers of the Support Level (Level 4/Phase 3) will replace/overwrite the contents of this function with their own code/implementation.

4 Exception Handling

As described above [Section 1], at startup, the Nucleus will have populated the Processor 0 Pass Up Vector with the address of the Nucleus exception handler (`exceptionHandler`) and the address of the Nucleus stack page (0x2000.1000). Therefore, if the Pass Up Vector was correctly initialized, `exceptionHandler` will be called (with a fresh stack) after each and every exception, exclusive of TLB-Refill events. Furthermore, the processor state at the time of the exception (the saved exception state) will have been stored (for Processor 0) at the start of the BIOS Data Page (0x0FFF.F000, constant BIOSDATAPAGE) [Section 3.2.2-pops].

The cause of this exception is encoded in the .ExcCode field of the Cause register (Cause.ExcCode) in the saved exception state [Section 3.3-pops].

Tip: To get the ExcCode you can use `getCAUSE()`, the constants `GETEXECCODE` and `CAUSESHIFT`.

- For exception code 0 (Interrupts, constant `IOINTERRUPTS`), processing should be passed along to your Nucleus's device interrupt handler [Section 8].
- For exception codes 1-3 (TLB exceptions), processing should be passed along to your Nucleus's TLB exception handler [Section 9.3].
- For exception codes 4-7, 9-12 (Program Traps), processing should be passed along to your Nucleus's Program Trap exception handler [Section 9.2].
- For exception code 8 (SYSCALL, constant `SYSEXCEPTION`), processing should be passed along to your Nucleus's SYSCALL exception handler [Section 5].

Important: On μ RISCV these code are different. Please, refer to Section 12 for μ RISCV's exception codes.

Hence, the entry point for the Nucleus's exception handling is in essence a case statement that performs a multi-way branch depending on the cause of the exception.

Important: To determine if the Current Process was executing in kernel-mode or user-mode, one examines the Status register in the saved exception state. In particular, examine the previous version of the KU bit (KUp) since the processor's exception handling circuitry will have performed a stack push on the KU/IE stacks in the Status register before the exception state was saved [Section 3.1-pops].

5 SYSCALL Exception Handling

A System Call (SYSCALL) exception occurs when the **SYSCALL** assembly instruction is executed. By convention, the executing process places appropriate values in the general purpose registers **a0-a3** immediately prior to executing the SYSCALL instruction. The Nucleus will then perform some service on behalf of the process executing the SYSCALL instruction depending on the value found in **a0**.

In particular, if the process making a SYSCALL request was in kernel-mode and **a0** contained a value in the range [-1..-2] then the Nucleus should perform one of the services described below.

5.1 SendMessage (SYS1)

This system call cause the transmission of a message to a specified process. This is an asynchronous operation (sender doesn't wait for receiver to perform a **ReceiveMessage** [Section 5.2]) and on success returns/places 0 in the caller's **v0**, otherwise **MSGNOGOOD** is used to provide a meaningful error condition on return. This system call doesn't require the process to lose its remaining time slice (it depends on scheduler policy). If the target process is in the **pcbFree_h** list, set the return register (**v0** in μ MPS3) to **DEST_NOT_EXIST**. If the target process is in the ready queue, this message is just pushed in its inbox, otherwise if the process is awaiting for a message, then it has to be awakened and put into the Ready Queue.

The SYS1 service is requested by the calling process by placing the value -1 in **a0**, the destination process PCB address in **a1**, the payload of the message in **a2** and then executing the **SYSCALL** instruction. The following C code can be used to request a SYS1:

```
SYSCALL(SENDMESSAGE, (unsigned int)destination, (unsigned int)payload, 0);
```

Where the mnemonic constant **SENDMESSAGE** has the value of -1.

5.2 ReceiveMessage (SYS2)

This system call is used by a process to extract a message from its inbox or, if this one is empty, to wait for a message. This is a synchronous operation since the requesting process will be frozen until a message matching the required characteristics doesn't arrive. This system call provides as returning value (placed in caller's **v0** in μ MPS3) the identifier of the process which sent the message extracted. This system call may cause the process to lose its remaining time slice, since if its inbox is empty it has to be frozen.

The SYS2 service is requested by the calling process by placing the value -2 in **a0**, the sender process PCB address or **ANYMESSAGE** in **a1**, a pointer to an area where the nucleus will store the payload of the message in **a2** (NULL if the payload should be ignored) and then executing the **SYSCALL** instruction. If **a1** contains a **ANYMESSAGE** pointer, then the requesting process is looking for the first message in its inbox, without any restriction about the sender. In this case it will be frozen only if the queue is empty, and the first message sent to it will wake up it and put it in the Ready Queue.

The following C code can be used to request a SYS2:

```
SYSCALL(RECEIVEMESSAGE, (unsigned int)sender, (unsigned int)payload, 0);
```

Where the mnemonic constant **RECEIVEMESSAGE** has the value of -2.

5.3 SYS1-SYS2 in User-Mode

The above two Nucleus services are considered privileged services and are only available to processes executing in kernel-mode. Any attempt to request one of these services while in user-mode should trigger a Program Trap exception response.

In particular the Nucleus should simulate a Program Trap exception when a privileged service is requested in user-mode. This is done by setting `Cause.ExcCode` in the stored exception state to `RI` (Reserved Instruction), and calling one's Program Trap exception handler.

Technical Point: As described above [Section 4], the saved exception state (for Processor 0) is stored at the start of the BIOS Data Page (`0xFFFF.F000`) [Section 3.2.2-pops].

5.4 Returning from a SYSCALL Exception

For SYSCALLs calls that do not block, control is returned to the Current Process at the conclusion of the Nucleus's SYSCALL exception handler. Observe that the correct processor state to load (`LDST`) is the saved exception state (located at the start of the BIOS Data Page [Section 4]) and not the obsolete processor state stored in the Current Process's PCB. The saved exception state was the state of the process at the time the SYSCALL was executed. The processor state in the Current Process's PCB was the state of the process at the start of its current time slice/quantum.

Hence, any return value described above needs to be put in the specified register in the stored exception state.

Furthermore, SYSCALLs that do not result in process termination (eventually) return control to the process's execution stream. This is done either immediately (e.g. `SYS1`) or after the process is blocked and eventually unblocked (e.g. `SYS2`). In any event the PC that was saved is, as it is for all exceptions, the address of the instruction that caused that exception – the address of the SYSCALL assembly instruction. Without intervention, returning control to the SYSCALL requesting process will result in an infinite loop of SYSCALL's. To avoid this, in case of `SYS1` or non-blocking `SYS2`, the PC must be incremented by 4 (i.e. the μ MPS3 wordsize, constant `WORDLEN`) prior to returning control to the interrupted execution stream. While the PC needs to be altered, there is no need, in this case, to make a parallel assignment to `t9`.

5.5 Blocking SYSCALLs

For SYSCALLs that block (`SYS2`), a number of steps need to be performed:

- The saved processor state (located at the start of the BIOS Data Page [Section 3]) must be copied into the Current Process's PCB (`p_s`).
- Update the accumulated CPU time for the Current Process [Section 10].
- Call the Scheduler.

6 System Service Interface

The System Service Interface (SSI) is a fundamental component of the kernel, since it provides services which are needed to build up higher levels of μ PandOS, such as process synchronization with I/O operation completion, pseudo-clock tick management, process proliferation (creation of children and siblings of processes), process termination and trap management processes specifications [Section 7].

The SSI runs inside the Kernel address space; each relevant system event will become a message managed by the SSI. The SSI will manage process requests on behalf of the nucleus; for example, a process which requires to know its accounted CPU time will send a message to the SSI, and wait for the answer. If the SSI ever gets terminated, the system must be stopped performing an emergency shutdown. The SSI process should implement the following RPC (remote procedure call) server algorithm:

```
while (TRUE) {  
    receive a request;
```

```

    satisfy the received request;
    send back the results;
}

```

The SSI could process the incoming request by using this function:

```
void SSIRequest(pcb_t* sender, int service, void* arg)
```

where **sender** will point to the area where the answer (if required) should be stored, **service** is a mnemonic code identifying the service requested and **arg** contains an argument (if required) for the service.

If **service** does not match any of those provided by the SSI, the SSI should terminate the process and its progeny. Also, when a process requires a service to the SSI, it must wait for the answer.

While **SSIRequest** has to be implemented depending on specific SSI implementation, these general issues have to be addressed:

- SSI requests should be implemented using SYSCALLs and message passing.
- There should be a way to identify requests addressed to SSI from other messages; the easiest way is to make the SSI process address available to requestor processes, but this could lead to a kernel security leak, so a magic number (a “fake” address recognized by the nucleus) could be used to address requests to SSI. This magic number must be chosen carefully, to avoid addressing ambiguities and/or address clashing with other processes (this feature is optional, therefore is not tested).
- SSI requests and answers could require more than one parameter; **msg_t** could be used in a creative way and/or expanded to allow the transport of “fat” messages.

7 Nucleus Services

These services should be implemented by the SSI on behalf of the nucleus, and could be requested by process running in Kernel mode. These services will be identified by mnemonic values.

7.1 CreateProcess

When requested, this service causes a new process, said to be a progeny of the sender, to be created. **arg** should contain a pointer to a **struct ssi_create_process_t** (**ssi_create_process_t ***). Inside this **struct**, **state** should contain a pointer to a processor state (**state_t ***). This processor state is to be used as the initial state for the newly created process. The process requesting the this service continues to exist and to execute. If the new process cannot be created due to lack of resources (e.g. no more free PCBs), an error code of -1 (constant **NOPROC**) will be returned, otherwise, the pointer to the new PCB will be returned. Good design calls for tight/strong cohesion and loose coupling between modules/classes/OS Levels, etc. Level 2 implements PCBs, and Level 3 utilizes queues of PCBs to create a basic multiprogramming environment. However, it is the Support Level that handles address translation as well as all exceptions beyond I/O interrupts and the first two system calls (and then, only if in kernel-mode). The design question then is how to provide Support Level access to PCB fields that will only be used in the Support Level. The standard approach, at least in systems-level programming such as an OS, is to define a structure containing the additional Support Level fields (**support_t**) and then add a pointer (**support_t ***) to the PCB. The Support Level code needing access to these fields will execute a **GetSupportData** [Section 7.6] which returns a pointer to the Current Process’s **support_t** structure. This provides Support Level access to relevant PCB fields while hiding the Level 3 (and Level 2) PCB fields.

This service is requested by the sender process by placing the value 1 in **service**, a pointer to the **struct ssi_create_process_t** **args**. This **struct** should contain a pointer to the process state in **state** and (optionally) a pointer to a Support Structure in **support**.

The mnemonic constant **CREATEPROCESS** has the value of 1.

The newly populated PCB is placed on the Ready Queue and is made a child of the Current Process. Process Count is incremented by one, and control is returned to the Current Process [Section 5.4]. In summary, one allocates a new PCB and initializes its fields:

- **p_s** from **arg->state**.
- **p_supportStruct** from **arg->support**. If no parameter is provided, this field is set to **NULL**.
- The process queue field (e.g. **p_list**) by the call to **insertProcQ**.
- The process tree field (e.g. **p_sib**) by the call to **insertChild**.
- **p_time** is set to zero; the new process has yet to accumulate any CPU time.

7.2 TerminateProcess

This services causes the sender process or another process to cease to exist [Section 11]. In addition, recursively, all progeny of that process are terminated as well. Execution of this instruction does not complete until all progeny are terminated.

The mnemonic constant **TERMINATEPROCESS** has the value of 2.

This service terminates the sender process if **arg** is **NULL**. Otherwise, **arg** should be a **pcb_t** pointer.

7.3 DoIO

μ PandOS supports only synchronous I/O; an I/O operation is initiated and the initiating process is blocked until the I/O completes. Whenever a process initiates an I/O operation, it will immediately issue a DoIO service call for that device. Hence, a DoIO will transit the Current Process from the “running” state to a “blocked” state.

The DoIO service is requested by sending a message to the SSI process with the following payload: the **ssi_payload_t**’s service code should be set to **DOIIO** and the **ssi_payload_t**’s argument should point to a **ssi_do_io_t** struct containing the command address and the command value. The SSI will eventually elaborate the service request. The I/O device requested will send a message to the SSI service when the I/O action is performed. Finally, the SSI will free the process waiting for the I/O action and will send to it the device response.

```
ssi_do_io_t do_io = {
    .commandAddr = command,
    .commandValue = value,
};
ssi_payload_t payload = {
    .service_code = DOIIO,
    .arg = &do_io,
};
// Send the request with the payload
SYSCALL(SENDMESSAGE, (unsigned int)ssi_pcb, (unsigned int>(&payload), 0);
// Wait for SSI’s response
SYSCALL(RECEIVEMESSAGE, (unsigned int) ssi_pcb, (unsigned int>(&response), 0);
```

Where the mnemonic constant `DOIIO` has the value of 3.

During this phase, only the print process will request the DoIO to the SSI process to write on the first terminal. Here is the step by step execution of the kernel when a generic DoIO is requested:

- A process sends a request to the SSI to perform a DoIO;
- the process will wait for a response from the SSI;
- the SSI will eventually execute on the CPU to perform the DoIO;
- given the device address, the SSI should save the waiting `pcb_t` on the corresponding device;
- at last, the SSI will write the requested value on the device, i.e. `*commandAddr = commandValue`;
Important: This call should rise an interrupt exception from a device;
- the SSI will continue its execution until it blocks again;
- every process should now be in a blocked state as everyone is waiting for a task to end, so the scheduler should call the `WAIT()` function; **Important:** The current process must be set to `NULL` and all interrupts must be `ON`;
- an interrupt exception should be raised by the CPU;
- given the cause code, the interrupt handler should understand which device triggered the `TRAP`;
- check the status and send to the device an acknowledge (setting the device command address to `ACK`);
- the device sends to the SSI a message with the status of the device operation, i.e. setting the `a3` parameter with the device address; now if the current process is `NULL`, return the control flow to the scheduler. Otherwise, load the current process state into the CPU;
- the kernel will execute the send, that will free the SSI process, that will elaborate the request from the device;
- given the device address, the SSI should free the process waiting the completion on the DoIO and finally, forwarding the status message to the original process.

Terminal devices are two independent sub-devices, and are handled by the `SYS5` service as two independent devices. As discussed below [Section 8], the SSI will send to the process a message containing the (sub)device's status. For character transmission and receipt, the status word, in addition to containing a device completion code, will also contain the character transmitted or received.

7.4 GetCPUTime

This service should allow the sender to get back the accumulated processor time (in μ seconds) used by the sender process. Hence, the Nucleus records (in the PCB: `p_time`) the amount of processor time used by each process [Section 10].

The mnemonic constant `GETTIME` has the value of 4.

7.5 WaitForClock

One of the services the nucleus has to implement is the pseudo-clock, that is, a virtual device which sends out an interrupt (a tick) every 100 milliseconds (constant `PSECOND`). This interrupt will be translated into a message to the SSI, as for other interrupts.

This service should allow the sender to suspend its execution until the next pseudo-clock tick. You need to save the list of PCBs waiting for the tick.

The mnemonic constant `CLOCKWAIT` has the value of 5.

7.6 GetSupportData

This service should allow the sender to obtain the process's Support Structure. Hence, this service returns the value of `p_supportStruct` from the sender process's PCB. If no value for `p_supportStruct` was provided for the sender process when it was created, return `NULL`.

The mnemonic constant `GETSUPPORTPTR` has the value of 6.

7.7 GetProcessID

This service should allow the sender to obtain the process identifier (PID) of the sender if argument is 0 or of the sender's parent otherwise. It should return 0 as the parent identifier of the root process.

The mnemonic constant `GETPROCESSID` has the value of 7.

8 Interrupt Exception Handling

Important: On μ RISCV, the codes below are different. Please, refer to Section 12 for μ RISCV's interrupt codes.

A device or timer interrupt occurs when either a previously initiated I/O request completes or when either a Processor Local Timer (PLT) or the Interval Timer makes a `0x0000.0000  $\Rightarrow$  0xFFFF.FFFF` transition.

Assuming that the (Processor 0) Pass Up Vector was properly initialized by the Nucleus as part of Nucleus initialization [Section 1], and that the Nucleus exception handler (`exceptionHandler`) correctly decodes `Cause.ExcCode` [Section 4], control should be passed to one's Nucleus interrupt exception handler.

Which interrupt lines have pending interrupts is set in `Cause.IP` [Section 3.3-pops]. Furthermore, for interrupt lines 3–7 the Interrupting Devices Bit Map will indicate which devices on each of these interrupt lines have a pending interrupt [Section 5.2.2-pops].

Tip: to figure out on which interrupt lines interrupts are pending, you can use a bitwise AND between `getCAUSE()` and the constants `...INTERRUPT`.

Since μ PandOS is intended for uniprocessor environments only, interrupt line 0 may safely be ignored [Chapter 5-pops].

Note, many devices per interrupt line may have an interrupt request pending, and that many interrupt lines may simultaneously be on. Also, since each terminal device is two sub-devices, each terminal device may have two interrupts pending simultaneously as well. One should process only one interrupt at a time: the interrupt with the highest priority. The lower the interrupt line and device number, the higher the priority of the interrupt. When there are multiple interrupts pending, and the interrupt exception handler processes only the single highest priority pending interrupt, the interrupt exception handler will be immediately re-entered as soon as interrupts are unmasked again; effectively forming a loop until all the pending interrupts are processed.

Since terminal devices are actually two sub-devices, both sub-devices may have an interrupt pending simultaneously. For purposes of prioritizing pending interrupts, terminal transmission (i.e. writing to the terminal) is of higher priority than terminal receipt (i.e. reading from the terminal). Hence, the PLT (interrupt line 1) is the highest priority interrupt, while reading from terminal 7 (interrupt line 7, device 7; read) is the lowest priority interrupt.

The interrupt exception handler's first step is to determine which device or timer with an outstanding interrupt is the highest priority.

Depending on the device, the interrupt exception handler will perform a number of tasks.

8.1 Non-Timer Interrupts

1. Calculate the address for this device's device register [Section 5.1-pops]:
 $\text{devAddrBase} = 0x10000054 + ((\text{IntlineNo} - 3) * 0x80) + (\text{DevNo} * 0x10)$
Tip: to calculate the device number you can use a `switch` among constants `DEVxON`.
2. Save off the status code from the device's device register.
3. Acknowledge the outstanding interrupt. This is accomplished by writing the acknowledge command code (`ACK`) in the interrupting device's device register. Alternatively, writing a new command in the interrupting device's device register will also acknowledge the interrupt.
4. Send a message and unblock the PCB waiting the status response from this (sub)device. This operation should unblock the process (PCB) which initiated this I/O operation and then requested the status response via a `SYS2` operation.
Important: Use of `SYS CALL` is discourage because both use `BIOSDATAPAGE`.
5. Place the stored off status code in the newly unblocked PCB's `v0` register.
6. Insert the newly unblocked PCB on the Ready Queue, transitioning this process from the "blocked" state to the "ready" state.
7. Return control to the Current Process: Perform a `LDST` on the saved exception state (located at the start of the BIOS Data Page [Section 4]).

Important: It is possible that there isn't any PCB waiting for this device. This can happen if while waiting for the initiated I/O operation to complete, an ancestor of this PCB was terminated. In this case, simply return control to the Current Process.

Important: It is also possible that there is no Current Process to return control to. This will be the case when the Scheduler executes the `WAIT` instruction instead of dispatching a process for execution [Section 2].

Technical Point: In μMPS3 it is technically feasible for a process to initiate an I/O operation and for the interrupt associated with this operation to occur before it has an opportunity to execute its `SYS2`. However, the μPandOS specification for the Support Level prevents this from happening.

8.2 Processor Local Timer (PLT) Interrupts

The PLT is used to support CPU scheduling. The Scheduler will load the PLT with the value of 5 milliseconds (constant `TIMESLICE`) whenever it dispatches a process [Section 2].

This "running" process will either:

- Terminate. Request a `TerminateProcess` to SSI or cause an exception without having set a Support Structure address [Section 9].
- Transition from the "running" state to the "blocked" state; execute a `SYS2`.
- Be interrupted by a PLT interrupt.

The last option means that the Current Process has used up its time quantum/slice but has not completed its CPU Burst. Hence, it must be transitioned from the "running" state to the "ready" state.

The PLT portion of the interrupt exception handler should therefore:

- Acknowledge the PLT interrupt by loading the timer with a new value [Section 4.1.4-pops].
- Copy the processor state at the time of the exception (located at the start of the BIOS Data Page [Section 3.2.2-pops]) into the Current Process's PCB (`p_s`).

- Place the Current Process on the Ready Queue; transitioning the Current Process from the “running” state to the “ready” state.
- Call the Scheduler.

8.3 The System-wide Interval Timer and the Pseudo-clock

The Pseudo-clock is a facility provided by the Nucleus for the Support Level. The Nucleus promises to unblock all the PCBs waiting for the Pseudo-clock [Section 1]. This periodic operation is called a Pseudo-clock Tick.

To wait for the next Pseudo-clock Tick (i.e. transition from the “running” state to the “blocked” state), the Current Process will request a `WaitForClock` service to the SSI [Section 7.5].

Since the Interval Timer is only used for this purpose, all line 2 interrupts indicate that it is time to unblock all PCBs waiting for a Pseudo-clock tick.

The Interval Timer portion of the interrupt exception handler should therefore:

1. Acknowledge the interrupt by loading the Interval Timer with a new value: 100 milliseconds (constant `PSECOND`) [Section 4.1.3-pops].
2. Unblock all PCBs blocked waiting a Pseudo-clock tick.
3. Return control to the Current Process: perform a `LDST` on the saved exception state (located at the start of the BIOS Data Page [Section 4]).

Important: It is also possible that there is no Current Process to return control to. This will be the case when the Scheduler executes the `WAIT` instruction instead of dispatching a process for execution [Section 2].

9 Pass Up or Die

The Nucleus will directly handle `SYS1-SYS2` requests and device (internal timers and peripheral devices) interrupts. For all other exceptions (e.g. `SYSCALL` exceptions numbered 1 and above, Program Trap and TLB exceptions) the Nucleus will take one of two actions depending on whether the offending process (i.e. the Current Process) was provided a non-NULL value for its Support Structure pointer when it was created [Section 7.1].

- If the Current Process’s `p_supportStruct` is NULL, then the exception should be handled as a `TerminateProcess`: the Current Process and all its progeny are terminated. This is the “die” portion of Pass Up or Die.
- If the Current Process’s `p_supportStruct` is non-NULL. The handling of the exception is “passed up”.

When an exception occurs, the processor, in concert with the BIOS-Excpt handler, “passes up” the handling of the exception to the Nucleus: store the saved exception state at an accessible location known to the Nucleus, and pass control to a routine specified by the Nucleus, i.e. the Nucleus Exception handler (`exceptionHandler`).

- The location, in this case, is fixed; a given location in the BIOS Data Page. (For Processor 0, this is `0x0FFF.F000`) [Section 3.2.2-pops]
- The address (and stack pointer) for the handler to pass control to was seeded by the Nucleus, during Nucleus initialization, in the appropriate location of the Pass Up Vector [Section 1].

When the Nucleus “passes up” exception handling to the Support Level, it essentially performs the same two tasks: copy the saved exception state into a location accessible to the Support Level, and pass control to a routine specified by the Support Level.

There is only one location for the saved exception state and one Pass Up Vector for the Nucleus. This is because the Nucleus runs in single threaded mode with interrupts masked; hence with no concurrency. The Nucleus services run in a “one at a time” mode, and each invocation running to completion without interruption. Hence the reusability of the BIOS Data Page location for the saved exception state and Pass Up Vector. This is also why Nucleus services are so limited: do only what must be done in single threaded mode, and pass up the handling of all other service requests.

Since the Support Level runs in a fully concurrent mode (interrupts unmasked), each process needs its own location(s) for their saved exception states, and addresses to pass control to: The Support Structure.

Furthermore, the concurrency at the Support Level is not only inter-process, but intra-process as well. The Support Level, while handling a passed up SYSCALL, can trigger a page fault. For this reason, the Support Structure contains two locations for saved exception states, and two addresses for handlers. One `state_t`/PC address pair for:

- TLB exceptions (i.e. page faults): The Support Level TLB exception handler.
- All other exceptions: The Support Level general exception handler.

One last important detail. The Support Structure’s version of a Pass Up Vector needs to contain three register values and not two. In addition to the PC/SP, one also needs a new value for the Status register.

A PC/SP/Status combination is also referred to as a context. Hence the Support Structure’s version of a Pass Up Vector needs to store two processor context sets: one for non-TLB exceptions and one for TLB exceptions.

The following two structures are provided:

```
/* process context */
typedef struct context_t {
    /* process context fields */
    unsigned int c_stackPtr, /* stack pointer value */
                c_status,    /* status reg value */
                c_pc;        /* PC address */
} context_t;

typedef struct support_t {
    int sup_asid;                /* Process Id (asid) */
    state_t sup_exceptState[2]; /* stored excpt states */
    context_t sup_exceptContext[2]; /* pass up contexts */
    // ... other fields to be added later
} support_t;

/* Exceptions related constants */
#define PGFAULTEXCEPT 0
#define GENERALEXCEPT 1
```

To pass up the handling of an exception:

- Copy the saved exception state from the BIOS Data Page to the correct `sup_exceptState` field of the Current Process. The Current Process’s PCB should point to a non-null `support_t`.
- Perform a LDCXT using the fields from the correct `sup_exceptContext` field of the Current Process [Section 7.3.4-pops].

9.1 SYSCALL Exceptions Numbered by positive numbers

A SYSCALL exception numbered 1 and above occurs when the Current Process executes the SYSCALL instruction (Cause.ExcCode is set to 8 [Section 4]) and the contents of `a0` is greater than or equal to 1.

The Nucleus SYSCALL exception handler should perform a standard Pass Up or Die operation using the `GENERALEXCEPT` index value.

9.2 Program Trap Exception Handling

A Program Trap exception occurs when the Current Process attempts to perform some illegal or undefined action. A Program Trap exception is defined as an exception with Cause.ExcCodes of 4-7, 9-12 [Section 4].

The Nucleus Program Trap exception handler should perform a standard Pass Up or Die operation using the `GENERALEXCEPT` index value.

9.3 TLB Exception Handling

A TLB exception occurs when μ MPS3 fails in an attempt to translate a logical address into its corresponding physical address. A TLB exception is defined as an exception with Cause.ExcCodes of 1-3 [Section 4].

The Nucleus TLB exception handler should perform a standard Pass Up or Die operation using the `PGFAULTEXCEPT` index value.

10 Accumulated CPU Time

μ MPS3 has three clocks: the TOD clock, Interval Timer, and the PLT, though only the Interval Timer and the PLT can generate interrupts. This fits nicely with two of three primary timing needs:

- Generate an interrupt to signal the end of Current Process's time quantum/slice. The PLT is reserved for this purpose.
- Generate Pseudo-clock ticks: Cause an interrupt to occur every 100 milliseconds and unblock all PCBs waiting for a Pseudo-clock tick. The Interval Timer is reserved for this purpose.

The third timing need is that the Nucleus is tasked with keeping track of the accumulated CPU time used by each process [Section 7.4].

A field has been defined in the PCB for this purpose (`p_time`). Hence `GetCPUTime` [Section 7.4] should return the value in the Current Process's `p_time` plus the amount of CPU time used during the current quantum/time slice. While the TOD clock does not generate interrupts, it is, however, well suited for keeping track of an interval's length.

By storing off the TOD clock's value at both the start and end of an interval, one can compute the duration of that interval [Section 4.1.2-pops].

The three timer devices are mechanisms for implementing μ PandOS's policies. Timing policy questions that need to be worked out include:

- While the time spent by the Nucleus handling an I/O or Interval Timer interrupt needs to be measured for Pseudo-clock tick purposes, which process, if any, should be "charged" with this time? Note: it is possible for an I/O or Interval Timer interrupt to occur even when there is no Current Process.
- While the time spent by the Nucleus handling a SYSCALL request needs to be measured for Pseudo-clock tick and quantum/time slice purposes, which process, if any, should be "charged" with this time?

It is important to understand the functional differences between the three μ MPS3 timer devices. This includes, but is not limited to understanding that the TOD clock counts up while the other two timers count down, and that the behavior of the PLT differs from that of the Interval Timer. The PLT can be enabled/disabled via the processor Local Timer enable bit (Status.TE) [Section 4.1.4-pops].

11 Process Termination

When a process is terminated (TerminateProcess or the “Die” portion of Pass Up or Die) there is actually a whole (sub)tree of processes that get terminated. There are a number of tasks that must be accomplished:

- The root of the sub-tree of terminated processes must be “orphaned” from its parents; its parent can no longer have this PCB as one of its progeny (outChild).
- If a terminated process is waiting for the completion of a DoIO, the value used to track this should be adjusted accordingly.
- If a terminated process is waiting for clock, the value used to track this should be adjusted accordingly.
- The process count and soft-blocked variables need to be adjusted accordingly.
- Processes (i.e. PCB’s) can’t hide. A PCB is either the Current Process (“running”), sitting on the Ready Queue (“ready”), blocked waiting for device (“blocked”), or blocked waiting for non-device (“blocked”).

12 Differences between μ MPS3 and μ RISCV

This is a summary of the main differences between μ MPS3 and μ RISCV. Full specifications and details can be found in the Dott. Rovelli’s thesis. The main differences are:

- registers; some register are not present or named differently. The table below shows only relevant differences:

μ MPS3	μ RISCV
v0	a0
reg_t9	not present
hi	not present
lo	not present

Table 1: Difference between registers

- in μ MPS3 and μ RISCV, if the program calls the HALT() function, “System halted” is written on the first terminal. But μ RISCV will hang the execution and you’ll need to force the exit of the program. If the program calls the PANIC() function, in μ MPS3, “kernel panic()” will be printed on the terminal; but, in μ RISCV, sometimes it will not print anything and it will just close the execution of the program.
- the status register is different (especially the user/machine mode section) and a new mie register is used for enabling interrupt (see Dott. Rovelli’s Section 2.1 for more details);
- interrupt and exceptions numbers are different. The table below shows only relevant differences:

Interrupt	Exception code	Description
1	3	Interval Timer
1	7	PLT Timer
1	17	Disk Device
1	18	Flash Device
1	18	Ethernet Device
1	20	Printer Device
1	21	Terminal Device
0	0-7, 12-23	Trap
0	8-11	Syscall
0	24-28	TLB exceptions

Table 2: μ RISCV exceptions codes

13 Nuts and Bolts

13.1 Module Decomposition

One possible module decomposition is as follows:

1. `initial.c` This module implements `main()` and exports the Nucleus's global variables (e.g. process count, soft blocked count, blocked PCBs lists/pointers, etc.).
2. `scheduler.c` This module implements the Scheduler and the deadlock detector.
3. `interrupts.c` This module implements the device/timer interrupt exception handler. This module will process all the device/timer interrupts, converting device/timer interrupts into appropriate messages for the blocked PCBs.
4. `exceptions.c` This module implements the TLB, Program Trap, and SYSCALL exception handlers. Furthermore, this module will contain the provided skeleton TLB-Refill event handler (e.g. `uTLB_RefillHandler`).
5. `ssi.c` This module implements the System Service Interface process.

13.2 Accessing the libumps Library

Accessing the CP0 registers and the BIOS-implemented services/instructions in C (e.g. `WAIT`, `LDST`) is via the `libumps` library [Chapter 7-pops].

Simply include the line

```
#include "/usr/include/umps3/umps/libumps.h"
```

in one's source files.¹

14 Testing

There is a provided test file, `p2test.c` that will “exercise” your code.

As with any non-trivial system, you are strongly encouraged to use the `make` program to maintain your code. A sample `Makefile` has been supplied. See Chapter 10 in the POPS reference for more compilation details.

¹The file `libumps.h` is part of the μ MPS3 distribution. `/usr/include/umps3/umps/` is the recommended installation location for this file.

Once your source files (from Phase 1 and Phase 2) have been correctly compiled, linked together (with appropriate linker script, `crtso.o`, and `libumps.o`), and post-processed with `umps3-elf2umps` (all performed by the sample `Makefile`), your code can be tested by launching the μ MPS3 emulator. At a terminal prompt, enter: `umps3`

Very Important: The `p2test.c` code assumes that the TLB Floor Address has been set to any value except VM OFF. The value of the TLB Floor Address is a user configurable value set via the μ MPS3 Machine Configuration Panel.

The test program reports on its progress by writing messages to `TERMINAL0`. At the conclusion of the test program, either successful or unsuccessful, μ MPS3 will display a final message and then enter an infinite loop. The final message will be **System Halted** for successful termination. We'll also evaluate your code implementation; in this phase you need to write relevant comments.

Good Luck!

μ PandOS: Phase 3

Luca Bassi (luca.bassi14@studio.unibo.it)
Gabriele Genovese (gabriele.genovese2@studio.unibo.it)

April 16, 2024

Phase 3 - Level 4: The Support Level

Level 4, the Support Level, builds on the Nucleus in two key ways to create an environment for the execution of user-processes (U-proc's):

1. Support for address translation/virtual memory. Each U-proc will execute in its own identically structured logical address space (kuseg), with a unique Address space identifier (i.e. process ID), ASID [Section 6.2-pops].
2. Support for character-oriented I/O devices: terminals and printers. Each U-proc is assigned its own printer and terminal.

Specifically, the Support Level provides the exception handlers that the Nucleus “passes” handling “up” to; assuming the process was provided a non-NULL value for its Support Structure.

There will be one Level 4/Phase 3 exception handler for:

1. TLB Management (TLB) exceptions: The Support Level page fault handler, i.e. the Pager [Section 4].
2. non-TLB exceptions. This handler is for all SYSCALL exceptions numbered 1 and above (positive numbers), and all Program Trap exceptions [Section 4.6].

These two exception handlers will run in kernel-mode with interrupts enabled, while the U-proc's will run in user-mode, with interrupts enabled. Hence each U-proc leads a schizophrenic life; mostly executing in user-mode, but sometimes, after the handling of an exception is “passed” back up to it; executing in kernel-mode. While the Nucleus exception and interrupt handlers are system-wide resources that all processes share (in serial fashion with interrupts disabled), the Support Level exception handlers are more like Support Level provided libraries that becomes part of each U-proc.¹ Finally, instead of using the Nucleus's test program (test) place holder TLB-Refill event handler (`uTLB_RefillHandler`), the Support Level will implement its own TLB-Refill event handler [Section 3]. Hence, the bulk of this phase is the implementation of these three exception event handlers.

Important: the following sections will refer to μ MPS3, but the differences with μ RISCV remain the same.

1 Address Translation: The OS Perspective

Before getting into how μ PandOS supports address translation, one must fully understand how the μ MPS3 hardware supports address translation [Chapter 6-pops] & [Figure 6.9-pops]. Essentially,

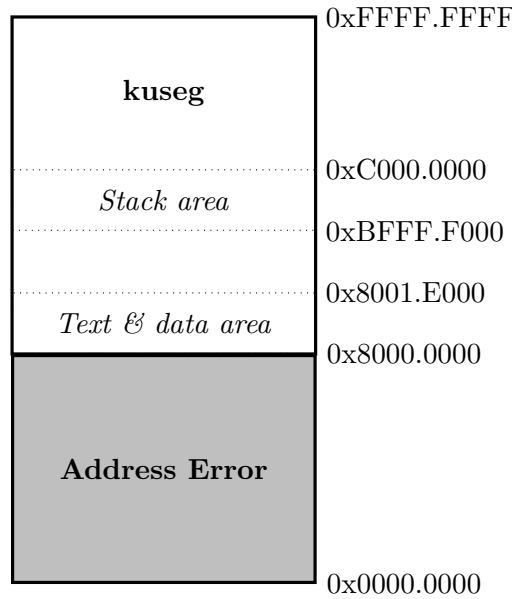
¹Technically, this is not true for the TLB-Refill event handler (e.g. `uTLB_RefillHandler`) which will behave like a Nucleus exception handler - a system-wide resource that all processes will share in serial fashion. However, since it is a part of the address translation process, it is included as part of Level 4/Phase 3.

every logical address for which translation is called for (any address above the TLB Floor Address) triggers a hardware search of the TLB seeking a matching TLB entry. If no matching entry is found a TLB-Refill event is triggered. Assuming the Nucleus correctly initialized the Processor 0 Pass Up Vector with the address of the TLB-Refill event handler [Section 3.1], control should continue with the Support Level's TLB-Refill event handler (e.g. `uTLB_RefillHandler`). This function will locate the correct Page Table entry in some Support Level data structure (i.e. a U-proc's Page Table), write it into the TLB (TLBWR or TLBWI [Section 6.4-pops] & [Section 5.2]), and return control (LDST) to the Current Process to restart the address translation process. Once a matching TLB entry is found and it is marked valid, the μ MPS3 hardware constructs the corresponding physical address. If the matching TLB entry is marked invalid, or the access represents an attempt to modify memory and the matching TLB entry's D bit is off, a TLB exception is raised: TLB-Invalid or TLB-Modification. The Support Level TLB exception handler will handle TLB-Invalid exceptions, i.e. page faults [Section 4]. Since all Page Table entries (and therefore all TLB entries) should be marked as dirty (the D bit on), TLB-Modification exceptions should not occur. This implies the following Support Level data structures:

- One Page Table per U-proc. A μ PandOS Page Table will be an array of 32 Page Table entries. Each Page Table entry is a doubleword consisting of an EntryHi and an EntryLo portion [Section 6.3.2-pops]. This array should be added to the Support Structure (`support_t`) that is pointed to by a U-proc's PCB.
Important: TLB entries and Page Table entries are identical in structure: a doubleword consisting of an EntryHi and an EntryLo portion. Which term is used will be dependent on context.
- The Swap Pool; a set of RAM frames reserved for virtual memory. Logical pages will occupy these frames when present. The size of the Swap Pool should be set to two times `UPROCMAX`, where `UPROCMAX` is defined as the specific degree of multiprogramming to be supported: [1...8]. The Swap Pool is not so much a Support Level data structure, but a set of RAM frames reserved to support paging.
- The Swap Pool data structure/table. The Support Level will maintain a table, one entry per Swap Pool frame, recording information about the logical page occupying it. At a minimum, each entry should record the ASID and logical page number of the occupying page.
- The Swap Mutex process. A process that provides mutual exclusion access to the Swap Pool table.
- Backing store; secondary storage that contains each U-proc's complete logical image – which for μ PandOS is limited to 32 pages in size. Associated with each U-proc is a flash device which will be configured (preloaded) to contain that U-proc's logical image. While slightly unrealistic, this basic version of the Support Level will use each U-proc's flash device as its backing store device.

2 A U-proc's Logical Address Space and Backing Store

Each U-proc executes in the kuseg address space [Section 6.2-pops], in user-mode, with interrupts enabled, and a unique ASID value. ASID 0 is reserved for kernel daemons, so the (up to) eight U-proc's should be assigned ASID values from [1..8]. The first page, for each U-proc is 0x8000.0000. The second page is 0x8000.1000, and so on. A μ PandOS U-proc's .text and .data regions, together can be no larger than 31 pages (0x8001.E000). The stack page is limited to one page and is set to the halfway point in kuseg. The SP will start at 0xC000.0000 and grow downward. μ PandOS, does not support dynamic variables, hence there is no heap space.



When a process is initiated, an operating system would typically read the contents of the executable file (e.g. `.aout` file) and use its contents to:

- Set up the new process's Page Table; which would reflect that none of the process's pages are present.
- Set up the new process's backing store on a secondary storage device.

2.1 A U-proc's Page Table

While the μ MPS3 hardware defines the structure of a TLB entry, it does not define the structure of a Page Table. A μ MPS3-compatible operating system is free to define a Page Table however it wishes; the hardware never interacts directly with Page Tables, just with the TLB. When a TLB-Refill event occurs, the operating system builds an appropriate TLB entry from the data in a Page Table and writes the entry into the TLB. To simplify this process, μ PandOS defines a Page Table entry to be identical to a TLB entry. Hence, in μ PandOS, a Page Table is an array of TLB entries. Each U-proc's Page Table will be an array of 32 TLB entries (or equivalently, an array of 32 Page Table entries). The first 31 entries are for the `.text` and `.data` pages of the logical address space (logical page number 0 through page number 30, starting from 0x8000.0000). The final entry is for the U-proc's stack page (logical page number 0x3FFF.F000, starting from 0x8000.0000).

	EntryHI		EntryLo				
	VPN	ASID	PFN	N	D	V	G
0	0x80000	<i>i</i>			1	0	
1	0x80001	<i>i</i>			1	0	
⋮	⋮	⋮			⋮	⋮	
30	0x8001E	<i>i</i>			1	0	
31	0xBFFFF	<i>i</i>			1	0	

To initialize a Page Table one needs to set the VPN, ASID, V, and D bit fields for each Page Table entry [Section 6.3.2-pops].

- The VPN field will be set to [0x80000..0x8001E] for the first 31 entries. The VPN for the stack page (Page Table entry 31) should be set to 0xBFFFF - the starting address whose top end is 0xC000.0000 (the value that SP is initialized to).

- The ASID field, for any given Page Table, will all be set to the U-proc's unique ID: an integer from [1..8].
- The D bit field will be set to 1 (on) - each page is write-enabled.
- The G bit field will be set to 1 (off) - these pages are private to the specific ASID.
- The V bit field will be set to 0 (off) - the entry is NOT valid. For example, a copy of this page is not also currently residing in RAM.

2.2 A U-proc's Backing Store

Since there is no file system (yet) containing files (executable or otherwise, e.g. `.aout`), which the operating system would read to set up both the Page Table and the backing store, the supplied utility `umps3-mkdev` [Section 11.2-pops] can be configured to preload a flash device with the contents of a `.aout` file in a manner that makes it suitable to be used as that process's backing store. Hence, user processes are not represented by a file to be processed (i.e. initialize a Page Table and set up the backing store), but via individual secondary storage devices (flash device) each preconfigured/already initialized with that process's logical image/backing store data. Specifically, each U-proc will be associated with a unique flash device, preloaded with that process's logical image, which the Support Level will then use as the process's backing store device.

3 The TLB-Refill event handler

When a logical address translation's search of the of the TLB for a matching entry fails, a TLB-Refill event is triggered. Assuming the Nucleus correctly initialized the Processor 0 Pass Up Vector with the address of the TLB-Refill event handler, control should continue with the μ PandOS TLB-Refill event handler (e.g. `uTLB_RefillHandler`). A TLB-Refill event is essentially a cache-miss event since the TLB is a cache of the most recently executed processes' Page Table entries. It is the job of the TLB-Refill event handler to insert into the TLB the missing Page Table entry and restart the instruction.

The Level 3/Phase 2 Nucleus code implemented a skeleton TLB-Refill event handler (e.g. `uTLB_RefillHandler`). The supplied skeleton code should, as part of this phase, be replaced (inplace) with the code for an actual TLB-Refill event handler.

Technical Point: The TLB-Refill event handler is actually a Level 3/Phase 2 handler in that it executes in kernel-mode, with interrupts disabled, and uses the first frame of RAM as its stack page; the Nucleus stack page. As such, like the other Level 3/Phase2 handlers (and unlike all the other Level 4/Phase 3 exception handlers) it is allowed access to the Level 3/Phase 2 global structures (e.g. Current Process). However, since it is a key component in μ PandOS's implementation of virtual memory, its implementation is part of Level 4/Phase 3, and therefore also has access to a process's Support Structure (e.g. the Page Table).

This function will:

- Locate the correct Page Table entry in the Current Process's Page Table; a component of `p_supportStruct`.
- Write the entry into the TLB using the `TLBWR` instruction [Section 6.4-pops].
- Return control (LDST) to the Current Process to restart the address translation process.

To accomplish this, a TLB-Refill event handler must:

1. Determine the page number (denoted as p) of the missing TLB entry by inspecting `EntryHi` in the saved exception state located at the start of the BIOS Data Page.

2. Get the Page Table entry for page number p for the Current Process. This will be located in the Current Process's Page Table, which is part of its Support Structure.
3. Write this Page Table entry into the TLB. This is a three-set process:
 - (a) `setENTRYHI`
 - (b) `setENTRYLO`
 - (c) `TLBWR`
4. Return control to the Current Process to retry the instruction that caused the TLB-Refill event: LDST on the saved exception state located at the start of the BIOS Data Page.

4 Paging in μ PandOS

4.1 The Swap Pool

A Swap Pool is a set of RAM frames set aside to support virtual memory. To ensure the proper exercise of μ PandOS's paging functionality, the size of the Swap Pool should be set to two times `UPROCMAX`, where `UPROCMAX` is defined as the specific degree of multiprogramming to be supported/implemented: [1...8] (i.e. the number of U-procs to be concurrently executed).

The Swap Pool can be placed anywhere in unused RAM: from the end of the operating system code, to the start of the last frame of RAM (which Level 3/Phase 2 allocated as the stack page for the initial process - test).

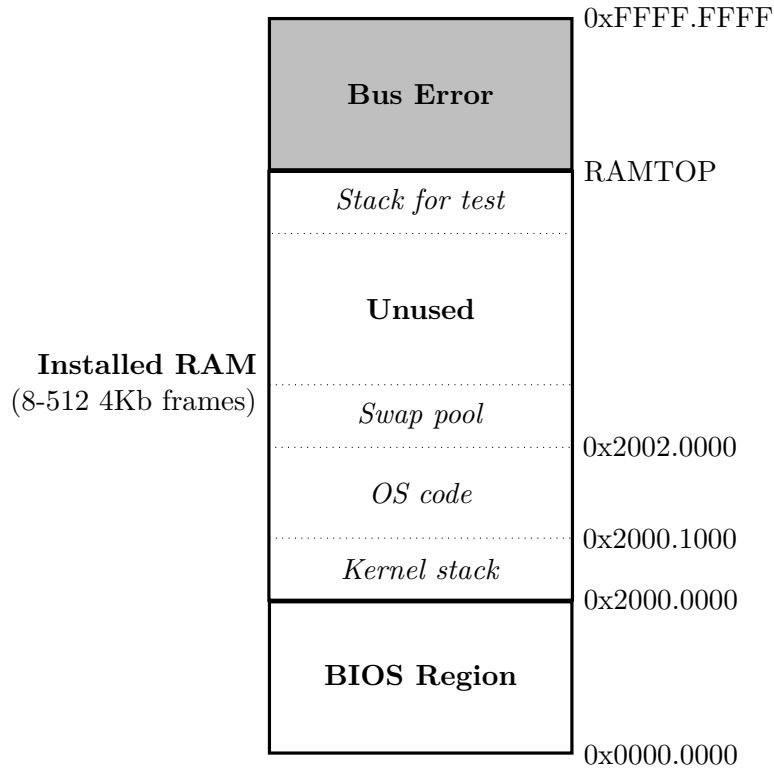
The recommended location in μ PandOS is to place the Swap Pool after the end of the operating system code. Though the size of one's operating system code is unknown², simply overestimate its size. For example, assume one's μ PandOS code base (plus Nucleus stack) occupies no more than 32 frames. Hence, the Swap Pool's starting address is: `0x20020000 (0x20000000 + (32 * PAGESIZE))`.

Important: Using the μ MPS3 Machine Configuration Panel make sure that there is sufficient "installed" RAM for the OS code, the Swap Pool and stack page for test [Section 12.2.1-pops].

The Support Level must maintain a table, one entry per frame in the Swap Pool, recording information about the logical page occupying it. This table should be composed of three columns/fields:

1. The ASID of the U-proc whose page is occupying the frame.
2. The logical page number (VPN) of the occupying page.
3. A pointer to the matching Page Table entry in the Page Table belonging to the owner process (i.e. ASID).

²The operating system object format, `.core` is a variant of the `.aout` format. The header information in both a `.core` and `.aout` file contains information describing the size of the code (`.text` and `.data`). [Section 10.3.1-pops]



Technical Point: Since all valid ASID values are positive numbers, one can indicate that a frame is unoccupied with an entry of -1 in that frame's ASID entry in the Swap Pool table.

The size of the table must match the size of the Swap Pool: one entry per frame in the Swap Pool.

Finally, the Swap Pool table is a shared data structure that must be accessed or updated in a mutually exclusive manner. Hence, the Support Level will also define a mutual exclusion process (the Swap Mutex process) to control access to the Swap Pool table.

To access the Swap Pool table, a process must first perform a message to this process and waiting for a response message. When access to the Swap Pool table is concluded, a process will then send a message to this process to release the mutual exclusion.

4.2 The Pager

While TLB-Refill events will be handled by the Support Level's TLB-Refill event handler (e.g. `uTLB_RefillHandler`), page faults are passed up by the Nucleus to the Support Level's TLB exception handler – the Pager.

μ MPS3 defines three different TLB exceptions [Chapter 3-pops]:

- Page fault on a load operation: TLB-Invalid exception – TLBL
- Page fault on a store operation: TLB-Invalid exception – TLBS
- An attempted write to a read-only page: TLB-Modification exception – Mod

In μ PandOS, Page Table entries are to be marked as read-writable, therefore TLB-Modification exceptions should not occur. If they do, they should be treated as a program trap [Section 9].

To handle a page fault, a μ PandOS TLB exception handler should perform the following steps:

1. Obtain the pointer to the Current Process's Support Structure requesting the `GetSupportData` service to the SSI. **Important:** Level 4/Phase 3 exception handlers are limited in their interaction with the Nucleus and its data structures to the functionality of SYSCALLs identified by negative numbers and service requests to the SSI.

2. Determine the cause of the TLB exception. The saved exception state responsible for this TLB exception should be found in the Current Process's Support Structure for TLB exceptions (`sup_exceptState[0]`'s Cause register).
3. If the Cause is a TLB-Modification exception, treat this exception as a program trap [Section 9], otherwise continue.
4. Gain mutual exclusion over the Swap Pool table sending a message to the `swap_table` PCB and waiting for a response.
5. Determine the missing page number (denoted as `p`): found in the saved exception state's `EntryHi`.
6. Pick a frame, `i`, from the Swap Pool. Which frame is selected is determined by the μ PandOS page replacement algorithm [Section 5.4].
7. Determine if frame `i` is occupied; examine entry `i` in the Swap Pool table.
8. If frame `i` is currently occupied, assume it is occupied by logical page number `k` belonging to process `x` (ASID) and that it is "dirty" (i.e. been modified):
 - (a) Update process `x`'s Page Table: mark Page Table entry `k` as not valid. This entry is easily accessible, since the Swap Pool table's entry `i` contains a pointer to this Page Table entry.
 - (b) Update the TLB, if needed. The TLB is a cache of the most recently executed process's Page Table entries. If process `x`'s page `k`'s Page Table entry is currently cached in the TLB it is clearly out of date; it was just updated in the previous step.
Important: This step and the previous step must be accomplished atomically [Section 5.3].
 - (c) Update process `x`'s backing store. Write the contents of frame `i` to the correct location on process `x`'s backing store/flash device [Section 5.1]. Treat any error status from the write operation as a program trap [Section 9].
9. Read the contents of the Current Process's backing store/flash device logical page `p` into frame `i` [Section 5.1]. Treat any error status from the read operation as a program trap [Section 9].
10. Update the Swap Pool table's entry `i` to reflect frame `i`'s new contents: page `p` belonging to the Current Process's ASID, and a pointer to the Current Process's Page Table entry for page `p`.
11. Update the Current Process's Page Table entry for page `p` to indicate it is now present (V bit) and occupying frame `i` (PFN field).
12. Update the TLB. The cached entry in the TLB for the Current Process's page `p` is clearly out of date; it was just updated in the previous step.
Important: This step and the previous step must be accomplished atomically [Section 5.3].
13. Release mutual exclusion over the Swap Pool table sending a message to the `swap_table` PCB.
14. Return control to the Current Process to retry the instruction that caused the page fault: `LDST` on the saved exception state.

5 Miscellaneous Details Related to Paging

5.1 Reading and Writing from/to a Flash Device

μ MPS3 flash devices are highly abstracted versions of real flash devices. It is convenient to think of them as isomorphic to seek-less, 1-dimensional disk devices. Flash device blocks are numbered sequentially [`0..MAXBLOCK-1`]. To read/write a flash device one performs the following two steps in order [Section 5.4-pops]:

1. Write the flash device's DATA0 field with the appropriate starting physical address of the 4k block to be read (or written); the particular frame's starting address.
2. Use the DoIO SSI service to write the flash device's COMMAND field with the device block number (high order three bytes) and the command to read (or write) in the lower order byte.

Each U-proc is associated with its own flash device, already initialized with its backing store data [Section 2.2]. The flash device's blocks [0..30] will be used to store the U-proc's .text, and .data, while block 31 will hold the U-proc's stack page.

5.2 Updating the TLB

The TLB is a cache of Page Table entries across multiple U-proc's. Hence, whenever a Page Table entry is updated by the Pager, if that entry is also present/cached in the TLB, there is a cache consistency problem. There are two approaches one can employ to guarantee cache consistency [Section 6.4-pops]. The two approaches are:

- Probe the TLB (TLBP) to see if the newly updated TLB entry is indeed cached in the TLB. If so (Index.P is 0), rewrite (update) that entry (TLBWI) to match the entry in the Page Table.
- Erase ALL the entries in the TLB (TLBCLR).

While the first approach is the recommended approach for μ PandOS. One should initially implement the second approach and then refactor to employ the first approach after all other aspects of the Support Level are completed/debugged.

5.3 Updating a Page Table and the TLB Atomically

The order of operations for the Pager are important. Specifically:

- When refreshing the backing store, one must first update the Page Table, and possibly the TLB, before performing the write operation.
- When reading in from the backing store, one must first perform the read operation before updating the Page Table and TLB.

Thought Challenge: Why must these operations be done in the prescribed order?

Similarly, the updating of a Page Table entry and its cached counterpart in the TLB, must be done atomically. This is accomplished in μ MPS3 by disabling interrupts before the update statements, and then reenabling them immediately afterwards. Interrupts are disabled and enabled via the STATUS register (`setStatus`) [Section 7.1-pops].

Thought Challenge: Why must the Page Table and TLB be updated atomically?

5.4 The μ PandOS Page Replacement Algorithm

When a page fault occurs, the page replacement algorithm picks one of the frames from the Swap Pool. The recommended μ PandOS page replacement algorithm is First in First out. Though inefficient, this "round robin" algorithm is easily implemented via a `static` variable. Whenever a frame is needed to support a page fault, simply increment this variable mod the size of the Swap Pool.

6 System Service Thread

The System Service Thread (SST) is a per-process thread that provide is child process useful services. Each SST child process can send a message to its SST to request a service.

SST create a child process that executes one of the test U-proc. SST shares the same ASID and support structure with its child U-proc. After its child initialization, the SST will wait for service requests from its child process (just like the SSI of the previous phase).

6.1 GetTOD

This service should allow the sender to get back the number of microseconds since the system was last booted/reset. The SST sends a message to the sender with the number of microseconds as payload.

```
ssi_payload_t sst_payload = {
    .service_code = GET_TOD,
    .arg = 0,
};
SYSCALL(SENDMSG, PARENT, (unsigned int)&sst_payload, 0);
```

The mnemonic constant `GET_TOD` has the value of 1.

6.2 Terminate

This service causes the sender U-proc and its SST (its parent) to cease to exist. It is essentially a SST “wrapper” for the SSI service `TerminateProcess`. Remember to send a message to the test process to communicate the termination of the SST.

```
ssi_payload_t sst_payload = {
    .service_code = TERMINATE,
    .arg = 0,
};
SYSCALL(SENDMSG, PARENT, (unsigned int)&sst_payload, 0);
```

The mnemonic constant `TERMINATE` has the value of 2.

6.3 WritePrinter

This service cause the print of a string of characters to the printer with the same number of the sender ASID. The sender must wait an empty response from the SST signaling the completion of the write.

```
sst_print_t print_payload = {
    .length = len,
    .string = str,
};
ssi_payload_t sst_payload = {
    .service_code = WRITEPRINTER,
    .arg = &print_payload,
};
SYSCALL(SENDMSG, PARENT, (unsigned int)&sst_payload, 0);
```

Where `str` is a `char*` to the string to write and `len` is its length.

The mnemonic constant `WRITEPRINTER` has the value of 3.

6.4 WriteTerminal

This service cause the print of a string of characters to the terminal with the same number of the sender ASID. The sender must wait an empty response from the SST signaling the completion of the write.

```
sst_print_t print_payload = {
    .length = len,
    .string = str,
};
ssi_payload_t sst_payload = {
    .service_code = WRITETERMINAL,
    .arg = &print_payload,
};
SYSCALL(SENDMSG, PARENT, (unsigned int)&sst_payload, 0);
```

Where `str` is a `char*` to the string to write and `len` is its length.

The mnemonic constant `WRITETERMINAL` has the value of 4.

7 The Support Level General Exception Handler

The Support Level general exception handler will process all passed up non-TLB exceptions:

- All SYSCALL (SYSCALL) exceptions numbered 1 and above (positive number).
- All Program Trap exceptions; all exception causes exclusive of those for SYSCALL exceptions and those related to TLB exceptions.

Assuming that the handling of the exception is to be passed up (non-NULL Support Structure pointer) and the appropriate `sup_exceptContext` fields of the Support Structure were correctly initialized, execution continues with the Support Level's general exception handler. The processor state at the time of the exception will be in the Support Structure's corresponding `sup_exceptState` field.

After examining the `sup_exceptState`'s Cause register, the Support Level general exception handler will pass control to either the Support Level's SYSCALL exception handler [Section 8], or the Support Level's Program Trap exception handler [Section 9].

8 The SYSCALL Exception Handler

The nucleus directly handles all Nucleus's SYSCALL exceptions (those having negative identifiers). For all other SYSCALL exceptions the nucleus either treats the exception as a process termination or "passes up" the handling of the exception if the offending process was provided a non-NULL value for its Support Structure pointer when it was created.

Assuming that the handling of the exception is to be passed up (non-NULL Support Structure pointer) and the appropriate `sup_exceptContext` fields of the Support Structure were correctly initialized, execution continues with the Support Level's general exception handler, which should then pass control to the Support Level's SYSCALL exception handler. The processor state at the time of the exception will be in the Support Structure's corresponding `sup_exceptState` field.

By convention the executing process places appropriate values in the general purpose registers `a0-a3` immediately prior to executing the SYSCALL instruction. The Support Level's SYSCALL exception handler will then perform some service on behalf of the U-proc executing the SYSCALL instruction depending on the value found in `a0`.

Upon successful completion of a SYSCALL request any return status is placed in `v0`, and control is returned to the calling process at the instruction immediately following the SYSCALL instruction. Similar to what the Nucleus does when returning from a successful SYSCALL request, the Support Level's SYSCALL exception handler must also increment the PC by 4 in order to return control to the instruction after the SYSCALL instruction.

In particular, if a U-proc executes a SYSCALL instruction and `a0` contained a valid positive value then the Support Level should perform one of the services described below.

8.1 SendMsg (USYS1)

This services cause the transmission of a message to a specified process. The USYS1 service is essentially a user-mode “wrapper” for the kernel-mode restricted SYS1 service. The USYS1 service is requested by the calling process by placing the value 1 in `a0`, the destination process PCB address or SST in `a1`, the payload of the message in `a2` and then executing the SYSCALL instruction. If `a1` contains PARENT, then the requesting process send the message to its SST [Section 6], that is its parent. The following C code can be used to request a USYS1:

```
SYSCALL(SENDMSG, (unsigned int)destination, (unsigned int)payload, 0);
```

Where the mnemonic constant `SENDMSG` has the value of 1.

8.2 ReceiveMsg (USYS2)

This system call is used by a process to extract a message from its inbox or, if this one is empty, to wait for a message. The USYS2 service is essentially a user-mode “wrapper” for the kernel-mode restricted SYS2 service. The USYS2 service is requested by the calling process by placing the value -2 in `a0`, the sender process PCB address or ANYMESSAGE in `a1`, a pointer to an area where the nucleus will store the payload of the message in `a2` (NULL if the payload should be ignored) and then executing the SYSCALL instruction. If `a1` contains a ANYMESSAGE pointer, then the requesting process is looking for the first message in its inbox, without any restriction about the sender. In this case it will be frozen only if the queue is empty, and the first message sent to it will wake up it and put it in the Ready Queue.

The following C code can be used to request a USYS2:

```
SYSCALL(RECEIVMSG, (unsigned int)sender, (unsigned int)payload, 0);
```

Where the mnemonic constant `RECEIVMSG` has the value of 2.

9 The Program Trap Exception Handler

For all Program Trap exceptions, the nucleus either treats the exception as a process termination or “passes up” the handling of the exception if the offending process was provided a non-NULL value for its Support Structure pointer when it was created.

Assuming that the handling of the exception is to be passed up (non-NULL Support Structure pointer) and the appropriate `sup_exceptContext` fields of the Support Structure were correctly initialized, execution continues with the Support Level's general exception handler, which should then pass control to the Support Level's Program Trap exception handler. The processor state at the time of the exception will be in the Support Structure's corresponding `sup_exceptState` field.

The Support Level's Program Trap exception handler is to terminate the process in an orderly fashion; perform the same operations as a SYS2 request.

Important: If the process to be terminated is currently holding mutual exclusion on a Support Level structure (e.g. Swap Pool table), mutual exclusion must first be released (send a message) before request the process termination.

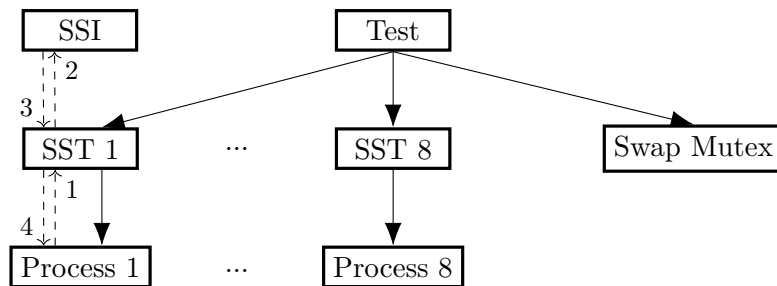
10 Process Initialization and test

The final step in Nucleus initialization is the instantiation of a single process (kernel-mode on, interrupts enabled) whose PC is set to test.

While test was the name/external reference to a function that exercised the Level 3/Phase 2 code, in Level 4/Phase 3 it will be used as the instantiator process (InstantiatorProcess).³

The InstantiatorProcess will perform the following tasks:

- Initialize the Level 4/Phase 3 data structures. These are:
 - The Swap Pool table and a Swap Mutex process that must provide mutex access to the swap table using message passing [Section 4.1].
 - Each (potentially) sharable peripheral I/O device can have a process for it. These process will be used to receive complex requests (i.e. to write of a string to a terminal) and request the correct DoIO service to the SSI (this feature is optional and can be delegated directly to the SST processes to simplify the project).
- Initialize and launch eight SST, one for each U-procs.
- Terminate after all of its U-proc “children” processes conclude. This will drive Process Count to one, triggering the Nucleus to invoke HALT. Wait for 8 messages, that should be send when each SST is terminated.



Technical Point: A careful reading of the Level 4/Phase 3 specification reveals that there are actually no purposefully shared peripheral devices. Each of the [1..8] U-procs has its own flash device (backing store), printer, and terminal device(s). Hence, one does not actually need a process to protect access to device registers. However, for purposes of correctness (or more appropriate: to protect against erroneous behavior) and future phase compatibility, it is possible to have a process for each device that waits for messages and requests the single DoIO to the SSI.

10.1 Initializing a U-proc

To launch a U-proc, one simply requests a CreateProcess to the SSI. The `ssi_create_process_t` that two parameters:

- A pointer to the initial processor state for the U-proc.
- A pointer to an initialized Support Structure for the U-proc.

Initial Processor State for a U-proc

Each U-proc’s initial processor state should have its:

- PC (and `s_t9`) set to 0x8000.00B0; the address of the start of the .text section [Section 10.3.1-pops].

³One is, of course, free to rename this function, however, that will entail going back and editing one’s already completed Level 3/Phase 2 code.

- SP set to 0xC000.0000 [Section 2].
- Status set for user-mode with all interrupts and the processor Local Timer enabled.
- EntryHi.ASID set to the process's unique ID; an integer from [1..8]

Important: Each U-proc MUST be assigned a unique, non-zero ASID.

10.1.1 Initialization of a Support Structure for a U-proc

Since the Support Level will launch and execute between 1 and 8 U-procs, there needs to be a pool of (up to) 8 Support Structures.

The recommended approach is to declare a static array of 8 Support Structures in `test`. Using an index variable (ASID?) one can easily obtain the address of the next unused Support Structure to be initialized and used for the next U-proc launch.

A Support Structure must contain all the fields necessary for the Support Level to support both paging and passed up SYSCALL services. This includes:

- `sup_asid`: The process's ASID.
- `sup_exceptState[2]`: The two processor state (`state_t`) areas where the processor state at the time of the exception is placed by the Nucleus for passing up exception handling to the Support Level.
- `sup_exceptContext[2]`: The two processor context (`context_t`) sets. Each context is a PC / SP / Status combination. These are the two processor contexts which the Nucleus uses for passing up exception handling to the Support Level.
- `sup_privatePgTbl[32]`: The process's Page Table.
- `sup_stackTLB[500]`: The stack area for the process's TLB exception handler. An integer array of 500 is a 2Kb area.
- `sup_stackGen[500]`: The stack area for the process's Support Level general exception handler.

Only the `sup_asid`, `sup_exceptContext[2]`, and `sup_privatePgTbl[32]` [Section 2.1] require initialization prior to request the CreateProcess service.

To initialize a processor context area one performs the following:

- Set the two PC fields. One of them (0 - `PGFAULTEXCEPT`) should be set to the address of the Support Level's TLB handler, while the other one (1 - `GENERALEXCEPT`) should be set to the address of the Support Level's general exception handler.
- Set the two Status registers to: kernel-mode with all interrupts and the Processor Local Timer enabled.
- Set the two SP fields to utilize the two stack spaces allocated in the Support Structure. Stacks grow "down" so set the SP fields to the address of the end of these areas.
E.g. `... = &(...sup_stackGen[499])`.

11 Small Support Level Optimizations

There are a number of small optimizations that one can undertake to improve the performance/organization of the Support Level.

In no particular order:

- Update the TLB by using TLBP and TLBWI instead of TLBCLR [Section 5.2].
- When a U-proc terminates, mark all of the frames it occupied as unoccupied [Section 4.1]. This has the potential to eliminate extraneous writes to the backing store.
- Improve the μ PandOS page replacement algorithm to first check for an unoccupied frame before selecting an occupied frame to use. This will turn an $O(1)$ operation into an $O(n)$ operation in exchange for fewer I/O (write) operations.
- Read each U-proc's header information and initialize the Page Table entries associated with each U-proc's .text pages as read only (D bit field set to 0/off) [Section 10.3.1-pops].
- Read μ PandOS's .core header information and situate the Swap Pool immediately after the .text and .data areas in RAM. This eliminates the need to overestimate the size of the operating system.

test cannot conclude before all of its spawned U-procs, otherwise, the Nucleus will prematurely terminate them [Section 10].

After launching all the U-procs, **test** should repeatedly wait for a message. This loop should iterate UPROC_{MAX} times: the number of U-proc's launched: [1..8].

Whenever a U-proc terminates, either normally, or abnormally, it should first send a message to the test PCB. Hence, **test** will go to sleep n times, and be woken up n times, where n is the number of launched U-procs ($n \in [1..8]$). After this loop concludes, **test** concludes by requesting a TerminateProcess to the SSI, which should trigger a HALT by the Nucleus.

- Allocate per-U-proc TLB, and general exception handler stacks directly from RAM [Section 10.1].

Directly allocate the two stack spaces per U-proc (one for the Support Level's TLB exception handler, and one for the Support Level's general exception handler) from RAM, instead of as fields in the Support Structure. The recommended RAM space to be used are the frames directly below RAMTOP, avoid the actual last frame of RAM (stack page for **test**).

Important: SP values are always the end of the area, not the start. Hence, to use the penultimate RAM frame as a U-proc's stack space for one of its Support Level handlers, one would assign the SP value to RAMTOP-PAGESIZE.

- Implement allocate and deallocate functionality for the Support Structures instead of directly accessing a static array [Section 10.1]. Instead of directly accessing elements of a static array of Support Structures, one can reuse the technique from Level 2/Phase 1: Declare a null-initialized pointer to a Support Structure-free list (stack?) of unused Support Structures. Upon entry, **test** iterates over the static array of Support Structures, invoking a new **deallocate** method to add each Support Structure to the free list. Whenever a new Support Structure is needed to support a new U-proc, a call to **allocate** returns a pointer to a Support Structure, allocated from the free list. Furthermore, whenever a U-proc terminates, a call is made to **deallocate** to return the Support Structure to the free list.

12 Nuts and Bolts

12.1 Initiating I/O Operations

A peripheral's device driver is typically made up of two parts: an upper part and a lower part.

The lower part is the code that handles the interrupt from the device upon completion of an operation. In μ PandOS this is handled by the Nucleus.

The upper part is the code that initiates an operation: the writing of some of the device's registers (except the COMMAND field) followed by a message send to SSI requesting the DoIO service (which sets the COMMAND field). In μ PandOS this code is distributed throughout the Support Level.

- For flash devices, the code to initiate reading and writing is part of (or at least called by) the Pager [Section 4.2].
- For printer devices the code is localized in the WritePrinter SST service implementation code [Section 6.3].
- For terminal devices the code is localized in the WriteTerminal SST service implementation code [Section 6.4].

12.2 Module Decomposition

One possible module decomposition is as follows:

1. `initProc.c`: This module implements test and exports the Support Level's global variables (e.g. swap mutex PCB [Section 10] and optionally the device PCBs).
2. `vmSupport.c`: This module implements the TLB exception handler (The Pager). Since reading and writing to each U-proc's flash device is limited to supporting paging, this module should also contain the function(s) for reading and writing flash devices. Additionally, the Swap Pool table is local to this module. Instead of declaring them globally in `initProc.c` they can be declared module-wide in `vmSupport.c`. The test function will now invoke a new "public" function `initSwapStructs` which will do the work of initializing the Swap Pool table.

Technical Point: Since the code for the TLB-Refill event handler was replaced (without relocating the function), `uTLB_RefillHandler` should still be found in the Level 3/Phase 2 `exceptions.c` file.

3. `sysSupport.c`: This module implements the Support Level's:
 - General exception handler [Section 7].
 - SYSCALL exception handler [Section 8].
 - Program Trap exception handler [Section 9].
4. `sst.c`: This module implements the System Service Thread:
 - Initialize the corresponding U-proc [Section 10.1].
 - Wait for service requests and manage them [Section 6].

12.3 Accessing the libumps Library

Accessing the CP0 registers and the BIOS-implemented services/instructions in C (e.g. `WAIT`, `LDST`) is via the `libumps` library [Chapter 7-pops].

Simply include the line

```
#include "/usr/include/umps3/umps/libumps.h"
```

in one's source files.⁴

⁴The file `libumps.h` is part of the μ MPS3 distribution. `/usr/include/umps3/umps/` is the recommended installation location for this file.

13 Testing

There is a provided set of possible U-proc programs that will “exercise” your code. These programs will generate page faults in addition to issuing SYSCALLs 1-2 and purposefully causing Program Traps.

The supplied U-proc programs also come with their own **Makefile** configured to compile, link (using the U-proc linker script, `crtsi.o`), create a corresponding flash device (a `.umps` file) [Section 11.2-pops], and preload the U-proc’s load image on to a flash device.

The recommended directory structure is to create a `testers` directory parallel to the other μ PandOS directories: `headers`, `phase1`, `phase2`, and `phase3`. As with any non-trivial system, you are strongly encouraged to use the `make` program to maintain your code. A sample **Makefile** has been supplied.

See Chapter 10 in the POPS reference for more compilation details. Once your source files (from Phase 1, Phase 2 and Phase 3) have been correctly compiled, linked together (with appropriate linker script, `crtso.o` and `libumps.o`), and post-processed with `umps3-elf2umps` (all performed by the sample **Makefile**), your code can be tested by launching the μ MPS3 emulator. At a terminal prompt, enter: `umps3`.

One uses the μ MPS3 Machine Configuration Panel [Section 12.2.1-pops] to set various parameters appropriate for testing μ PandOS:

- The TLB Floor Address must be set to either 0x4000.0000 or 0x8000.0000.
- The amount of “installed” RAM must be sufficiently large enough for the OS code, the Swap Pool and stack page for `test` (e.g. 128 frames).
- Using the Devices tab one maps a flash device (`.umps`) “file” with the corresponding μ MPS3 flash device [Section 12.2.1-pops]. Simply use the Browse button to locate the appropriate `.umps` file (in the `testers` directory) and enable the device via the checkbox.