# MultiPandOS: Phase 3

Luca Bassi (luca.bassi14@studio.unibo.it)
Luca Orlandello (luca.orlandello@studio.unibo.it)

April 10, 2025

## 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.

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 6].

These two exception handlers will run in kernel-mode with interrupts disable, 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.

## 1 Address Translation: The OS Perspective

Before getting into how MultiPandOS supports address translation, one must fully understand how the $\mu$RISC-V hardware supports address translation. Essentially, 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.

[1] Technically, 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.

Assuming the Nucleus correctly initialized the Pass Up Vector of each processor with the address of the TLB-Refill event handler, 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 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 $\mu$RISC-V 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 MultiPandOS 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. 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.

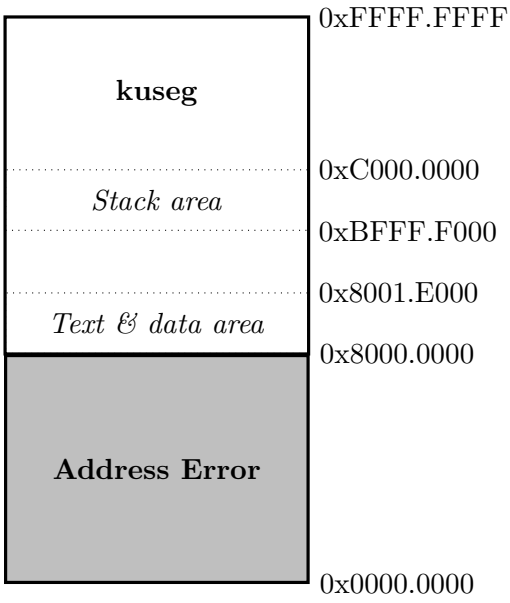
- The Swap Pool data structure/table. The Support Level will maintain a table, one entry per Swap Pool frame, recording information about the logical page occupying it. At a minimum, each entry should record the ASID and logical page number of the occupying page.

- The Swap Mutex process. A process that provide mutual exclusion access to the Swap Pool table.

- Backing store; secondary storage that contains each U-proc's complete logical image – which for MultiPandOS 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, 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 MultiPandOS 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. MultiPandOS, 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 $\mu$RISC-V hardware defines the structure of a TLB entry, it does not define the structure of a Page Table. A $\mu$RISC-V-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 simply this process, MultiPandOS defines a Page Table entry to be identical to a TLB entry. Hence, in MultiPandOS, 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.

| | EntryHI | | EntryLo | | | | |
|---|---|---|---|---|---|---|---|
| | VPN | ASID | PFN | N | D | V | G |
| 0 | 0x80000 | *i* | | | 1 | 0 | |
| 1 | 0x80001 | *i* | | | 1 | 0 | |
| ⋮ | ⋮ | ⋮ | | | ⋮ | ⋮ | |
| 30 | 0x8001E | *i* | | | 1 | 0 | |
| 31 | 0xBFFFF | *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.

- 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 0 (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 `uriscv-mkdev` 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 TLB for a matching entry fails, a TLB-Refill event is triggered. Assuming the Nucleus correctly initialized the Pass Up Vector of each processors with the address of the TLB-Refill event handler, control should continue with the MultiPandOS 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.

Each processor has its own different and independent TLB. When a U-proc is executing on a processor, the logical address translation searches for a matching entry in the TLB of the current processor. If it fails, the TLB-Refill event handler needs to update the TLB of the current processor.

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 MultiPandOS'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.
- 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 of the CPU (you can use `getPRID()` to get the processor ID).

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 of the CPU.

# 4 Paging in MultiPandOS

## 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 MultiPandOS'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 MultiPandOS is to place the Swap Pool after the end of the operating system code and processors kernel stacks. Though the size of one's operating system code is unknown[2], simply overestimate its size. Hence, the Swap Pool's starting address is:
`RAMSTART + (64 * PAGESIZE) + (NCPU * PAGESIZE)`

**Important**: Using the $\mu$RISC-V Machine Configuration Panel make sure that there is sufficient "installed" RAM for the OS code, the Swap Pool and stack page for `test` (e.g. 512 frames).

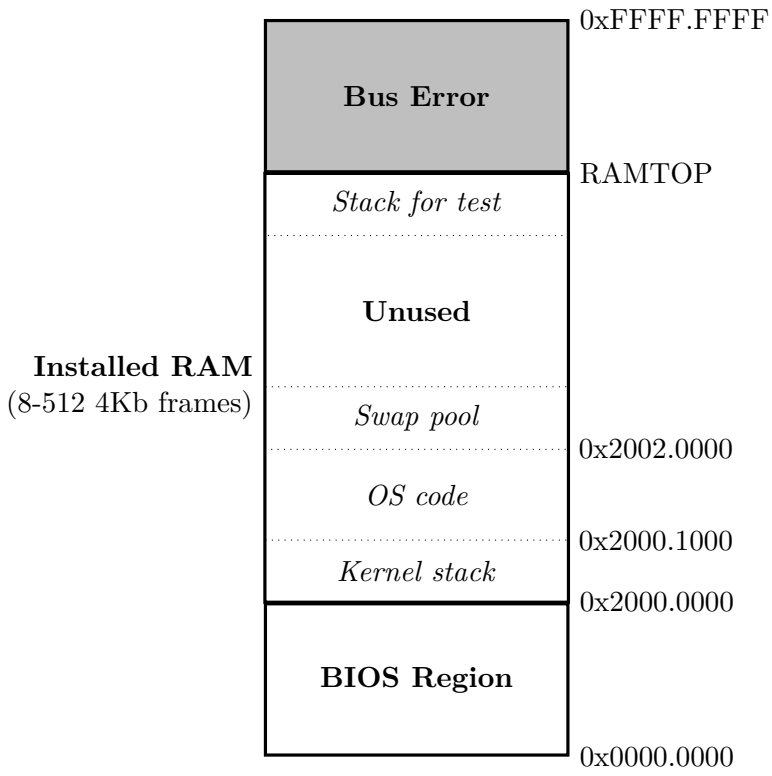
The Support Level must maintain a table, one entry per frame in the Swap Pool, recording information about the logical page occupying it. This table should be composed of three columns/fields:

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).

[2]The 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).




**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 semaphore (the Swap Pool semaphore) to control access to the Swap Pool table.

To access the Swap Pool table, a process must first perform a NSYS3 (P) operation on this semaphore. When access to the Swap Pool table is concluded, a process will then perform a NSYS4 (V) operation on this semaphore. Since this semaphore is used for mutual exclusion, it should be initialized to one.

### 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.

$\mu$RISC-V defines three different TLB exceptions:

- 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 MultiPandOS, 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 8].

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

1. Obtain the pointer to the Current Process's Support Structure: NSYS8.
   **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.

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 8], otherwise continue.

4. Gain mutual exclusion over the Swap Pool table (NSYS3 – P operation on the Swap Pool semaphore).

5. Determine the missing page number (denoted as `p`): found in the saved exception state's EntryHi.

6. Check if the page is loaded in the swap pool but the TLB of the current processor is not updated. In particular:

   (a) Update the TLB, if needed. The TLB is a cache of the most recently executed process's Page Table entries. If current process's page `p`'s Page Table entry is currently cached in the TLB it can be outdated.

   (b) Update the value in the TLB.

   (c) If the page is valid: release mutual exclusion over the Swap Pool table (NSYS4 – V operation on the Swap Pool semaphore) and return control to the Current Process to retry the instruction that caused the page fault: `LDST` on the saved exception state.

   **Technical Point**: This can happen if another processor has loaded the page because each CPU has a separated TLB.

7. Pick a frame, `i`, from the Swap Pool. Which frame is selected is determined by the MultiPandOS page replacement algorithm [Section 5.4].

8. Determine if frame `i` is occupied; examine entry `i` in the Swap Pool table.

9. 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 8].

10. 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 8].

11. 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`.

12. 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).

13. 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].

14. Release mutual exclusion over the Swap Pool table (NSYS4 – V operation on the Swap Pool semaphore).

15. 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

$\mu$RISC-V 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:

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 NSYS5 system call 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 used 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. 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`).

The `TLBP` (TLB-Probe) command initiates a TLB search for a matching entry in the TLB that matches the current values in the EntryHi register, one can set the EntryHi register with `setENTRYHI`. If a matching entry is found in the TLB the corresponding index value is loaded into `Index.TLB-Index` and the Probe bit (`Index.P`) is set to 0. If no match is found, `Index.P` is set to 1. One can obtain the content of the `Index` register with `getINDEX`. One can use `PRESENTFLAG` to check `Index.P` bit.

The `TLBCLR` (TLB-Clear) command zero's out the "unsafe" TLB entries; entries 1 through `TLBSIZE`-1. This command effectively invalidates the current contents of the TLB cache.

When a `TLBWI` instruction is executed, the contents of the EntryHi and EntryLo registers are written into the slot indicated by `Index.TLB-Index`. One can set the EntryLo register with `setENTRYLO`.

When a `TLBWR` instruction is executed, the contents of the EntryHi and EntryLo registers are written into a random slot.

While the first approach is the recommended approach for MultiPandOS. 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 $\mu$RISC-V by disabling interrupts before the update statements, and then reenabling them immediately afterwards. Interrupts are disabled and enabled via the STATUS register (setSTATUS).

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

### 5.4 The MultiPandOS Page Replacement Algorithm

When a page fault occurs, the page replacement algorithm picks one of the frames from the Swap Pool. The recommended MultiPandOS 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 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 namely all exception causes except 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 hander will pass control to either the Support Level's SYSCALL exception handler [Section 7], or the Support Level's Program Trap exception handler [Section 8].

## 7 The SYSCALL Exception Handler

The nucleus directly handles all NSYS SYSCALL exceptions (those having negative identifiers). For all other SYSCALL exceptions the nucleus either treats the exception as a NSYS2 (terminate) 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 `a0`, 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.

### 7.1 Terminate (SYS2)

This services causes the executing U-proc to cease to exist. The SYS2 service is essentially a user-mode "wrapper" for the kernel-mode restricted NSYS2 service.

The SYS2 service is requested by the calling process by placing the value `2` in `a0` and then executing a SYSCALL instruction.

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

```
SYSCALL(TERMINATE, 0, 0, 0);
```

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

### 7.2 WritePrinter (SYS3)

When requested, this service causes the requesting U-proc to be suspended until a line of output (string of characters) has been transmitted to the printer device associated with the U-proc.

Once the process resumes, the number of characters actually transmitted is returned in `a0`.

The SYS3 service is requested by the calling U-proc by placing the value `3` in `a0`, the virtual address of the first character of the string to be transmitted in `a1`, the length of this string in `a2`, and then executing a SYSCALL instruction. Once the process resumes, the number of characters actually transmitted is returned in `a0` if the write was successful. If the operation ends with a status other than "Device Ready" (1), the negative of the device's status value is returned in `a0`.

It is an error to write to a printer device from an address outside of the requesting U-proc's logical address space, request a SYS3 with a length less than 0, or a length greater than 128. Any of these errors should result in the U-proc being terminated (SYS2).

The following C code can be used to request a SYS3:

```
int retValue = SYSCALL(WRITEPRINTER, char *virtAddr, int len, 0);
```

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

### 7.3 WriteTerminal (SYS4)

When requested, this service causes the requesting U-proc to be suspended until a line of output (string of characters) has been transmitted to the terminal device associated with the U-proc.

The SYS4 service is requested by the calling U-proc by placing the value `4` in `a0`, the virtual address of the first character of the string to be transmitted in `a1`, the length of this string in `a2`, and then executing a SYSCALL instruction. Once the process resumes, the number of characters actually transmitted is returned in `a0` if the write was successful. If the operation ends with a status other than "Character Transmitted" (`5`), the negative of the device's status value is returned in `a0`.

It is an error to write to a terminal device from an address outside of the requesting U-proc's logical address space, request a SYS4 with a length less than 0, or a length greater than 128. Any of these errors should result in the U-proc being terminated (SYS2).

The following C code can be used to request a SYS4:

```
int retValue = SYSCALL(WRITETERMINAL, char *virtAddr, int len, 0);
```

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

### 7.4 ReadTerminal (SYS5)

When requested, this service causes the requesting U-proc to be suspended until a line of input (string of characters) has been transmitted from the terminal device associated with the U-proc.

The SYS5 service is requested by the calling U-proc by placing the value 5 in `a0`, the virtual address of a string buffer where the data read should be placed in `a1`, and then executing a SYSCALL instruction. Once the process resumes, the number of characters actually transmitted is returned in `a0` if the read was successful. If the operation ends with a status other than "Character Received" (5), the negative of the device's status value is returned in `a0`.

Attempting to read from a terminal device to an address outside of the requesting U-proc's logical address space is an error and should result in the U-proc being terminated (SYS2).

The following C code can be used to request a SYS5:

```
int retValue = SYSCALL(READTERMINAL, char *virtAddr, 0, 0);
```

Where the mnemonic constant `READTERMINAL` has the value of 5.

## 8 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 semaphore (e.g. Swap Pool semaphore), mutual exclusion must first be released (NSYS4) before invoking the Nucleus terminate command (NSYS2).

## 9 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:

[3] One 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.

- Initialize the Level 4/Phase 3 data structures. These are:
  - The Swap Pool table and Swap Pool semaphore [Section 4.1].
  - Each (potentially) sharable peripheral I/O device should have a semaphore defined for it. These semaphores will be used for mutual exclusion (protect access to each device's device registers) and therefore should all be initialized to one. Since terminal devices are actually two independent sub-devices, each terminal device should have two mutual exclusion semaphores defined for it: one for reading from the terminal and one for writing to the terminal.
- Initialize and launch (NSYS1) between 1 and 8 U-procs.
- Terminate (NSYS2) after all of its U-proc "children" processes conclude. This will drive Process Count to zero, triggering the Nucleus to invoke `HALT`.

**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 an array of mutual exclusion semaphores to protect access to device registers. However, for purposes of correctness (or more appropriate: to protect against erroneous behaviour) and future phase compatibility, it is strongly recommended one define and use this array of mutual exclusion device register semaphores.

## 9.1 Initializing a U-proc

To launch a U-proc, one simply sets up the parameters for a NSYS1, followed by the actual execution of the NSYS1 Nucleus service.

The NSYS1 Nucleus service takes two parameters:

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

### 9.1.1 Initial Processor State for a U-proc

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

- PC set to 0x8000.00B0; the address of the start of the .text section.
- 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.

### 9.1.2 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.
- 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])`.

# 10 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 MultiPandOS 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.
- Introduce a `masterSemaphore` for a more graceful conclusion/termination of `test`.
  `test` cannot conclude before all of its spawned U-procs, otherwise, the Nucleus will prematurely terminate them.
  Introduce a new Support Level-level semaphore; the zero-initialized `masterSemaphore`. After launching all the U-procs, test should repeatedly issue a NSYS3 (V operation) on this semaphore. This loop should iterate `UPROCMAX` times: the number of U-proc's launched: [1..8].
  Whenever a U-proc terminates, either normally, or abnormally, it should first perform a NSYS4 (V operation) on the `masterSemaphore`. 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 issuing a NSYS2, which should trigger a HALT by the Nucleus.

# 11 Nuts and Bolts

## 11.1 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 MultiPandOS 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 NSYS5 (which sets the COMMAND field). In MultiPandOS 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 SYS3 implementation code [Section 7.2].
- For terminal devices the code is localized in the SYS4 & SYS5 implementation code [Section 7.3 & 7.4].

## 11.2 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. device semaphores [Section 9], and optionally a `masterSemaphore` [Section 10].
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 6].
   - SYSCALL exception handler [Section 7].
   - Program Trap exception handler [Section 8].

## 11.3 Accessing the `liburiscv` Library

Accessing the CP0 registers and the BIOS-implemented services/instructions in C (e.g. `WAIT`, `LDST`) is via the `liburiscv` library.

Simply include the line

```
#include <uriscv/liburiscv.h>
```

in one's source files.[4]

[4]The file `liburiscv.h` is part of the $\mu$RISC-V distribution. `/usr/include/uriscv/` is the recommended installation location for this file.

# 12 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 `.uriscv` file), 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 Multi-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 `liburiscv.o`), and post-processed with `uriscv-elf2uriscv` (all performed by the sample `Makefile`), your code can be tested by launching the $\mu$RISC-V emulator. At a terminal prompt, enter: `uriscv`.

One uses the $\mu$RISC-V Machine Configuration Panel to set various parameters appropriate for testing MultiPandOS:

- 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 (`.uriscv`) "file" with the corresponding $\mu$RISC-V flash device. Simply use the Browse button to locate the appropriate `.uriscv` file (in the `testers` directory) and enable the device via the checkbox.
- Set the number of the processor to 8 (remember to keep this number aligned with the constant `NCPU`). It's easier and suggested starting with 1 processor and only after increasing the number.

**Important**: Before calling `HALT` in the scheduler, one needs to add this snippet of code because the BIOS expects that the interrupts arrives at the processor that calls `HALT`.

```
unsigned int *irt_entry = (unsigned int*) IRT_START;
for (int i = 0; i < IRT_NUM_ENTRY; i++) {
    *irt_entry = getPRID();
    irt_entry++;
}
```