# $\mu$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).

$\mu$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 $\mu$MPS3 and $\mu$RISCV. In this document reference will be made to an implementation for $\mu$MPS3. For the differences between $\mu$MPS3 and $\mu$RISCV, see section 12.

# 1 Nucleus Initialization

Every program needs an entry point (i.e. `main()`). The entry point for $\mu$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_refll_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 $\mu$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. $\mu$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. $\mu$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 $\mu$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*) 0x0FFFF000);
}
```

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 $\mu$RISCV these code are different. Please, refer to Section 12 for $\mu$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 $\mu$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 $\mu$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 (0x0FFF.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 $\mu$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 $\mu$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 processes 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 PBCs), 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

$\mu$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 `DOIO` 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 = DOIO,
    .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 `DOIO` 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 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 $\mu$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 $\mu$RISCV, the codes below are different. Please, refer to Section 12 for $\mu$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 $\mu$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]:
   `devAddrBase = 0x10000054 + ((IntlineNo - 3) * 0x80) + (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 `SYSCALL` 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 $\mu$MPS3 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 $\mu$PandOS 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 $\mu$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

$\mu$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 $\mu$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 $\mu$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 $\mu$MPS3 and $\mu$RISCV

This is a summary of the main differences between $\mu$MPS3 and $\mu$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:

| **$\mu$MPS3** | **$\mu$RISCV** |
|---|---|
| `v0` | `a0` |
| `reg_t9` | not present |
| `hi` | not present |
| `lo` | not present |

Table 1: Difference between registers

- in $\mu$MPS3 and $\mu$RISCV, if the program calls the HALT() function, "System halted" is written on the first terminal. But $\mu$RISCV will hang the execution and you'll need to force the exit of the program. If the program calls the PANIC() function, in $\mu$MPS3, "kernel panic()" will be printed on the terminal; but, in $\mu$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 | Termimal Device |
| 0 | 0-7, 12-23 | Trap |
| 0 | 8-11 | Syscall |
| 0 | 24-28 | TLB exceptions |

Table 2: $\mu$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.

[1]The file `libumps.h` is part of the $\mu$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 µMPS3 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 $\mu$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, $\mu$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!