

PandOSsh: Phase 2

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

December 10, 2025

1 Phase 2 - Level 3: The Nucleus

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

1. Receives control from the exception handling facility of Level 1. There are two categories of exceptions:
 - 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, and the facility to handle both system service calls and device interrupts, timer interrupts in particular, provide a process scheduler – support multiprogramming and multiprocessor.

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.

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 PCB is placed on the ASL and the Scheduler is called to dispatch the next job. If the system call is non-blocking, control is returned to the Current Process.
- It terminates; which is signaled via a system call. 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 control is returned to the Current Process.

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

- Nucleus initialization [Section 2].
- The Scheduler [Section 3].
- Exception handling and SYSCALL processing [Section 6].
- Device interrupt handler [Section 7].
- The passing up of the handling of all other events. This includes TLB-Refill events [Section 4], SYSCALLs not handled at this level, page faults, Program Trap exceptions, etc. [Section 8].

2 Nucleus Initialization

Every program needs an entry point (i.e. `main()`). The entry point for PandOSsh 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 are in the “blocked” state due to an I/O or timer request.
 - Ready Queue: 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.
 - Device Semaphores: The Nucleus maintains one integer semaphore for each external (sub)device, plus one additional semaphore to support the Pseudo-clock. Since terminal devices are actually two independent sub-devices, the Nucleus maintains two semaphores for each terminal device.
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 [Section 11]. The Pass Up Vector is where the BIOS finds the address of the Nucleus functions to pass control to for both TLB-Refill events and all other exceptions. Specifically,
 - Set the Nucleus TLB-Refill event handler address to


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

 where `memaddr`, in types.h, has been aliased to `unsigned int`. Since address translation is not implemented until the Support Level, `uTLB_RefillHandler` is a place holder function whose code is provided [Section 4]. This code will then be replaced when the Support Level is implemented.

- Set the Stack Pointer for the Nucleus TLB-Refill event handler to the top of the Nucleus stack page: 0x2000.1000 (constant KERNELSTACK). Stacks in μ RISC-V 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 5]:

```
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();
initASL();
```

4. Initialize all the previously declared variables: Process Count (0), Soft-block Count (0), Ready Queue (`mkEmptyProcQ()`), and Current Process (NULL). Since the device semaphores will be used for synchronization, as opposed to mutual exclusion, they should all be initialized to zero.
5. Load the system-wide Interval Timer with 100 milliseconds (constant PSECOND) [Section 7.3].
6. Instantiate a single 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 `test`. To enable interrupt set the mie field of the processor state inside the PCB to the constant `MIE_ALL`. To enable interrupt and kernel mode set the status field of the processor state inside the PCB to the constant `MSTATUS_MPIE_MASK | MSTATUS_MPP_M` see Dott. Rovelli's thesis for more details. 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 blocking semaphore address (`p_semAdd`) to NULL.
- Set the Support Structure pointer (`p_supportStruct`) to NULL.

`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

```
pcb->p_s.pc_epc = (memaddr) test;
```

Remember to declare `test` as “external” in your program by including the line:

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

7. 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 for Processor 0. 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 (stack grows down), will now be some value less, due to the activation record for `main()` that now sits on the stack.

3 The Scheduler

Your Nucleus should guarantee finite progress; consequently, every ready process will have an opportunity to execute. The Nucleus should implement a simple preemptive round-robin scheduling algorithm with a time slice value of 5 milliseconds (constant `TIMESLICE`). Preemptive CPU scheduling requires the use of an interrupt generating system clock. μ RISC-V offers two choices: the single system-wide Interval Timer or a Processor’s Local Timer (PLT). One should use the PLT to support per processor scheduling since the Interval Timer is reserved for implementing Pseudo-clock ticks [Section 7.3]. 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 7.2].
3. Perform a Load Processor State (LDST) [Section 13.2] on the processor state stored in PCB of the Current Process (`p_s`) of the current CPU.

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 0, invoke the HALT BIOS service/instruction [Section 13.2]. Consider this a job well done!
2. If the Process Count > 0 and Soft-block Count > 0 enter a Wait State. A Wait State is where the processor is not executing instructions, but “twiddling its thumbs” waiting for a device interrupt to occur. μ RISC-V supports a WAIT instruction expressly for this purpose [Section 13.2].

Important: Before executing the WAIT instruction, the Scheduler must first set the mie register to enable interrupts and either disable the PLT (also through the mie register) using:

```
setMIE(MIE_ALL & ~MIE_MTIE_MASK);
unsigned int status = getSTATUS();
status |= MSTATUS_MIE_MASK;
setSTATUS(status);
```

see Dott. Rovelli’s thesis for more details.

The first interrupt that occurs after entering a Wait State should not be for the PLT.

3. Deadlock for PandOSsh is defined as when the Process Count > 0 and the Soft-block Count is zero. Take an appropriate deadlock detected action; invoke the PANIC BIOS service/instruction [Section 13.2].

4 TLB-Refill events

As outlined above [Section 2], each Processor 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() {
    int prid = getPRID();
    setENTRYHI(0x80000000);
    setENTRYLO(0x00000000);
    TLBWR();
    LDST((state_t*) BIOSDATAPAGE);
}
```

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

5 Exception Handling

As described above [Section 2], at startup, the Nucleus will have populated the Processor 0 Pass Up Vectors with the address of the Nucleus exception handler (`exceptionHandler`) and the address of the Nucleus stack page. 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 (0xFFFF.F000, constant `BIOSDATAPAGE`). You can use the `GET_EXCEPTION_STATE_PTR(id)` macro to access the BIOS Data Page [Section 11] of the various CPUs.

The cause of this exception is encoded in the cause register in the saved exception state.

Tip: To get the exception code you can use `getCAUSE()` [Section 13.2], the macro `CAUSE_IS_INT`, and the constant `CAUSE_EXCCODE_MASK`.

- If `CAUSE_IS_INT`¹ is true (the current exception is an interrupt), processing should be passed along to your Nucleus's device interrupt handler [Section 7];
- For exception codes 24-28 (TLB exceptions), processing should be passed along to your Nucleus's TLB exception handler [Section 8.3];
- For exception codes 8 and 11, processing should be passed along to your Nucleus's SYSCALL exception handler [Section 6];
- For other exception codes 0-7, 9, 10, 12-23 (Program Traps), processing should be passed along to your Nucleus's Program Trap exception handler [Section 8.2].

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 MPP field with `saved_exception_state->status & MSTATUS_MPP_MASK` (see section Dott. Rovelli's thesis for more details).

6 SYSCALL Exception Handling

A System Call 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 negative range then the Nucleus should perform one of the services described below. See `/usr/include/uriscv/types.h` on how access to registers `a0-a3` from the saved exception state of the current processor.

6.1 CreateProcess (NSYS1)

When requested, this service causes a new process, said to be a progeny of the caller, to be created. `a1` should contain a pointer to a processor state (`state_t *`). This processor state is to be used as

¹The `CAUSE_IS_INT` macro checks if the most significant bit of the parameter is 1, if the most significant bit of the cause register is 1 it means that the exception was caused by an interrupt.

the initial state for the newly created process. The process requesting the NSYS1 service continues to exist and to execute. If the new process cannot be created due to lack of resources (e.g. no more free PCBs), an error code of -1 is placed/returned in the caller's **a0**, otherwise, return the process id of the newly created process in the caller's **a0**. 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 eight 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 NSYS8 [Section 6.8] 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. The NSYS1 service is requested by the calling process by placing the value -1 in **a0**, a pointer to a processor state in **a1**, a process priority value in **a2**, (optionally) a pointer to a Support Structure in **a3**, and then executing the SYSCALL instruction.

The following C code can be used to request a NSYS1:

```
int retValue = SYSCALL(CREATEPROCESS, state_t *statep, int prio, support_t *supportp);
```

Where 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.

In summary, for NSYS1, one allocates a new PCB and initializes its fields:

- **p_s** from **a1**;
- **p_supportStruct** from **a3**. If no parameter is provided, this field is set to NULL.
- A new value must be generated and assigned to **p_pid**.
- The process queue fields (e.g. **p_next**) by the call to **insertProcQ**.
- The process tree fields (e.g. **p_child**) by the call to **insertChild**.
- Update process count.
- **p_time** is set to zero; the new process has yet to accumulate any CPU time.
- **p_semAdd** is set to NULL; this PCB/process is in the “ready” state, not the “blocked” state.

6.2 TerminateProcess (NSYS2)

This services causes the executing process or another process to cease to exist. In addition, recursively, all progeny of that process are terminated as well. Execution of this instruction does not complete until all progeny are terminated, after which the Scheduler should be called. The NSYS2 service is requested by the calling process by placing the value -2 in **a0** and then executing the SYSCALL instruction.

The following C code can be used to request a NSYS2:

```
SYSCALL(TERMINATEPROCESS, int pid, 0, 0);
```

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

This service terminates the calling process if PID is zero, the process whose identifier is PID otherwise.

6.3 Passeren (P) (NSYS3)

This service requests the Nucleus to perform a P operation on a semaphore. The P or NSYS3 service is requested by the calling process by placing the value -3 in **a0**, the physical address of the semaphore to be P'ed in **a1**, and then executing the SYSCALL instruction. Depending on the value of the semaphore, the value of the semaphore is decreased and control is returned to the Current Process, or this process is blocked on the ASL (transitions from “running” to “blocked”) and the Scheduler is called.

The following C code can be used to request a NSYS3:

```
SYSCALL(PASSEREN, int *semaddr, 0, 0);
```

Where the mnemonic constant PASSEREN has the value of -3.

6.4 Verhogen (V) (NSYS4)

This service requests the Nucleus to perform a V operation on a semaphore. The V or NSYS4 service is requested by the calling process by placing the value -4 in **a0**, the physical address of the semaphore to be V'ed in **a1**, and then executing the SYSCALL instruction. The V operation is non blocking, depending on the value of the semaphore, control is either returned to the Current Process, or the semaphore value is increased.

The following C code can be used to request a NSYS4:

```
SYSCALL(VERHOGEN, int *semaddr, 0, 0);
```

Where the mnemonic constant VERHOGEN has the value of -4.

6.5 DoIO (NSYS5)

PandOSsh supports only synchronous I/O; an I/O operation is initiated, and the initiating process is blocked until the I/O completes. In order to begin an I/O operation a process should assign a value to the Command field of the device register. NSYS5 assign a value to that command field for that device [Section 12]. Hence, a NSYS5 is used to transition the Current Process from the “running” state to a “blocked” state. More formally, this service performs a P operation on the semaphore that the Nucleus maintains for the I/O device indicated by the value in **a1**. Since the semaphore that will have a P operation performed on it is a synchronization semaphore, this call should always block the Current Process on the ASL, after which the Scheduler is called. Terminal devices are two independent sub-devices, and are handled by the NSYS5 service as two independent devices. Hence each terminal device has two Nucleus maintained semaphores for it; one for character receipt and one for character transmission. As discussed below [Section 7], the Nucleus will perform a V operation on the Nucleus maintained semaphore whenever that (sub)device generates an interrupt. Once the process resumes after the occurrence of the anticipated interrupt, the (sub)device’s status word is returned in **a0**. For character transmission and receipt, the status word, in addition to containing a device completion code, will also contain the character transmitted or received. The NSYS5 service is requested by the calling process by placing the value -5 in **a0**, the address of the command field in **a1** and the value to be assigned in **a2**.

The following C code can be used to request a NSYS5:

```
int ioStatus = SYSCALL(DOIO, int *commandAddr, int commandValue, 0);
```

Where the mnemonic constant DOIO has the value of -5.

6.6 GetCPUTime (NSYS6)

This service requests that the accumulated processor time (in microseconds) used by the requesting process be placed/returned in the caller's a0. Hence, the Nucleus records (in the PCB: p_time) the amount of processor time used by each process [Section 9]. The NSYS6 service is requested by the calling process by placing the value -6 in a0 and then executing the SYSCALL instruction.

The following C code can be used to request a NSYS6:

```
cpu_t cpuTime = SYSCALL(GETCPUTIME, 0, 0, 0);
```

Where the mnemonic constant GETCPUTIME has the value of -6.

6.7 WaitForClock (NSYS7)

This service performs a P operation on the Nucleus maintained Pseudo-clock semaphore. This semaphore is V'ed every 100 milliseconds by the Nucleus [Section 7.3]. Since the Pseudo-clock semaphore is a synchronization semaphore, this call should always block the Current Process on the ASL, after which the Scheduler is called. Hence, a NSYS7 is used to transition the Current Process from the "running" state to a "blocked" state. The NSYS7 service is requested by the calling process by placing the value -7 in a0 and then executing the SYSCALL instruction.

The following C code can be used to request a NSYS7:

```
SYSCALL(WAITCLOCK, 0, 0, 0);
```

Where the mnemonic constant WAITCLOCK has the value of -7.

6.8 GetSupportData (NSYS8)

This service requests a pointer to the Current Process's Support Structure. Hence, this service returns the value of p_supportStruct from the Current Process's PCB. If no value for p_supportStruct was provided for the Current Process when it was created, return NULL. The NSYS8 service is requested by the calling process by placing the value -8 in a0 and then executing the SYSCALL instruction.

The following C code can be used to request a NSYS8:

```
support_t *sPtr = SYSCALL(GETSUPPORTPTR, 0, 0, 0);
```

Where the mnemonic constant GETSUPPORTPTR has the value of -8.

6.9 GetProcessID (NSYS9)

The NSYS9 service is requested by the calling process by placing the value -9 in a0 and then executing the SYSCALL instruction. The following C code can be used to request a NSYS9:

```
int pid or ppid = SYSCALL(GETPID, int parent, 0, 0);
```

Where the mnemonic constant GETPID has the value of -9.

The process id (PID) of the calling process is placed/returned in the caller's a0 if parent is zero. The process id of the parent process (PID) of the calling process is placed/returned in the caller's a0 otherwise. (It should return zero as the parent identifier of the root process)

6.10 Yield (NSYS10)

This service causes the calling process to relinquish the CPU. If there are other processes in the ready queue, the yielded process is not immediately re-executed even if it has the highest priority. If it is the only ready process, it is executed. The NSYS10 service is requested by the calling process by placing the value -10 in `a0` and then executing the SYSCALL instruction. The following C code can be used to request a NSYS10:

```
SYSCALL(YIELD, 0, 0, 0);
```

Where the mnemonic constant `YIELD` has the value of -10.

6.11 NSYS1-NSY10 in User-Mode

The SYSCALL identified by negative values are Nucleus services. These 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. Any attempt to request a non-existent Nucleus service should trigger a Program Trap exception too.

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

6.12 Returning from a SYSCALL Exception

For SYSCALLs calls that do not block or terminate, 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 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 (e.g. NSYS6) 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. NSYS6) or after the process is blocked and eventually unblocked (e.g. NSYS5). 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 the PC must be incremented by 4 (i.e. the μ RISC-V wordsize) prior to returning control to the interrupted execution stream.

6.13 Blocking SYSCALLs

For SYSCALLs that block (NSYS3, NSYS5, NSYS7 and NSYS10), a number of steps need to be performed:

- As described above [Section 6.12] the value of the PC must be incremented by 4 to avoid an infinite loop of SYSCALLs.
- The saved processor state must be copied into the Current Process's PCB (`p_s`).
- Update the accumulated CPU time for the Current Process [Section 9].
- In case of NSYS5, write the command to the device register.
- Call the Scheduler.

7 Interrupt Exception Handling

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 2], and that the Nucleus exception handler (`exceptionHandler`) correctly decodes that the current exception is an interrupt using `CAUSE_IS_INT` [Section 5], control should be passed to one's Nucleus interrupt exception handler. Depending on the interrupt exception code we update a local variable `IntlineNo` according to the Table 1. The interrupt line 1 is for Process Local Timer (PLT), the line 2 is for Pseudo-clock, and the other lines are for devices. For interrupt lines 3–7 the Interrupting Devices Bit Map will indicate which devices on each of these interrupt lines have a pending interrupt.

The Interrupting Devices Bit Map is a read-only five word area located starting from address $0x1000.0040$. Interrupting Devices Bit Map words have this format: when bit i in word j is set to one then device i attached to interrupt line $j + 3$ has a pending interrupt, see Table 2. An interrupt pending bit is turned on automatically by the hardware whenever a device's controller asserts the interrupt line to which it is attached. The interrupt will remain pending –the pending interrupt bit will remain on– until the interrupt is acknowledged. Interrupts for peripheral devices are acknowledged by writing the acknowledge command code in the appropriate device's device register.

Device Class	Macro Name	Interrupt Exception Code	IntlineNo
Process Local Timer	<code>IL_CPUTIMER</code>	7	1
Interval Timer	<code>IL_TIMER</code>	3	2
Disk Devices	<code>IL_DISK</code>	17	3
Flash Devices	<code>IL_FLASH</code>	18	4
Network (Ethernet) Devices	<code>IL_ETHERNET</code>	19	5
Printer Devices	<code>IL_PRINTER</code>	20	6
Terminal Devices	<code>IL_TERMINAL</code>	21	7

Table 1: Interrupt Line and Device Class Mapping

Word #	Physical Address	Field Name
4	$0x1000.0040 + 0x10$	Interrupt Line 7 Interrupting Devices Bit Map
3	$0x1000.0040 + 0x0C$	Interrupt Line 6 Interrupting Devices Bit Map
2	$0x1000.0040 + 0x08$	Interrupt Line 5 Interrupting Devices Bit Map
1	$0x1000.0040 + 0x04$	Interrupt Line 4 Interrupting Devices Bit Map
0	$0x1000.0040$	Interrupt Line 3 Interrupting Devices Bit Map

Table 2: Interrupting Devices Bit Map Addresses

Tip: to figure out the current interrupt exception code, you can use a bitwise AND between `getCAUSE()` and the constants `CAUSE_EXCCODE_MASK`.

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 interrupt line and device, the interrupt exception handler will perform a number of tasks.

7.1 Non-Timer Interrupts

1. Calculate the address for this device's device register [Section 12]:

Given an interrupt line (`IntLineNo`) and a device number (`DevNo`) one can compute the starting address of the device's device register:

```
devAddrBase = 0x10000054 + ((IntlineNo - 3) * 0x80) + (DevNo * 0x10)
```

Tip: to calculate the device number you can use a series of `ifs` with bitwise AND (`&`) between the bitmap and the `DEVxON` constants as conditions.

2. Save off the status code from the (sub)device's device register.
3. Acknowledge the outstanding interrupt. This is accomplished by writing the acknowledge command code (constant `ACK`) in the interrupting (sub)device's command register. Alternatively, writing a new command in the interrupting (sub)device's device register will also acknowledge the interrupt.
4. Perform a V operation on the Nucleus maintained semaphore associated with this (sub)device. This operation should unblock the process (PCB) which initiated this I/O operation and then requested to wait for its completion.
5. Place the stored off status code in the newly unblocked PCB's `a0` 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 if exists: perform a LDST on the saved exception state of the current CPU, otherwise call the scheduler.

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 3].

7.2 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 3].

This "running" process will either:

- Terminate. Execute a `NSYS2` or cause an exception without having set a Support Structure address.
- Transition from the "running" state to the "blocked" state; execute a `NSYS3`, `NSYS5`, `NSYS7`.

- 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 using `setTIMER` [Section 13.2].
- Copy the processor state of the current CPU at the time of the exception into the Current Process’s PCB (`p_s`) of the current CPU.
- Place the Current Process on the Ready Queue; transitioning the Current Process from the “running” state to the “ready” state.
- Call the Scheduler.

7.3 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 2]. 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), a process will request a `WaitForClock` nucleus syscall [Section 6.7]. 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`). Load the Interval Timer value can be done with the following pre-defined macro `LDIT(T)`.
2. Unblock all PCBs blocked waiting a Pseudo-clock tick.
3. Return control to the Current Process if exists: perform a `LDST` [Section 13.2] on the saved exception state of the current CPU.

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` [Section 13.2] instruction instead of dispatching a process for execution [Section 3].

8 Pass Up or Die

The Nucleus will directly handle all NSYS requests (negative numbered) 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 6.1].

- If the Current Process’s `p_supportStruct` is NULL, then the exception should be handled as a `TerminateProcess`: the Current Process of the current CPU 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-Exception 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 the saved exception state.
- 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 2].

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.

Each process has its own location(s) for their saved exception states, and addresses to pass control to: The Support Structure.

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 of the current CPU from the BIOS Data Page to the correct `sup_exceptState` field of the Current Process of the current CPU. The Current Process's PCB should point to a non-null `support_t`.
- Perform a LDCXT [Section 13.2] using the fields from the correct `sup_exceptContext` field of the Current Process of the current CPU.

8.1 SYSCALL Exceptions Numbered by positive numbers

A SYSCALL exception numbered 1 and above occurs when the Current Process of the current CPU executes the SYSCALL instruction 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.

8.2 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 codes 0-7, 9, 10, 12-23 [Section 5].

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

8.3 TLB Exception Handling

A TLB exception occurs when μ RISC-V fails in an attempt to translate a logical address into its corresponding physical address. A TLB exception is defined as an exception with codes 24-28 [Section 5].

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

9 Accumulated CPU Time

μ RISC-V has three clocks: the Time Of Day (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 6.6].

A field has been defined in the PCB for this purpose (`p_time`). Hence GetCPUTime [Section 6.6] 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. It is set to zero at system boot/reset time and begins counting up by one after each clock cycle. 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. Access to the TOD clock value can be accomplished with the following pre-defined macro `STCK(T)`.

```
cpu_t current_time;
STCK(current_time);
```

The three timer devices are mechanisms for implementing PandOSsh 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 PandOSsh 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.

10 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`).
- 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”).

11 BIOS Data page

When an exception occurs during normal execution, the BIOS handler routines store off the processor state at the time of the exception on the BIOS Data Page at a location accessible to the kernel. One processor state is 37 words long and there can be up to 16 processors/cores. Starting at 0xFFFF.F000 are eight, 37-word areas. The exception processor state for exceptions associated with Processor 0 is found at 0xFFFF.F000. The exception processor state for exceptions associated with processor 1 is found at 0xFFFF.F000 + 0x94 (i.e. 0x94 is the size of a processor state: $37 * 4 = 148 = 0x94$). The exception processor state for exceptions associated with processor 2 is found at 0xFFFF.F000+ 0x128, etc. The second action performed by both the BIOS handler routines is to pass control to the kernel. Since there are both exceptions and TLB-Refill events there needs to be two addresses for each processor. While the BIOS is careful not to use/need a stack, the same cannot be said for the kernel. Hence in addition to two (PC) addresses (for each processor), there also needs to be two SP values: one for each handler, which might be the same for both handlers. The BIOS Data Page in addition to providing space for 16 processor states, also provides space for 16 four-word areas. Each four word area, known as Pass Up Vector) is defined as in Table 3.

The Pass Up Vector for Processor 0 is located at 0xFFFF.F900. The Pass Up Vector Processor 1 is located at 0xFFFF.F900 + 0x10. The Pass Up Vector for Processor 2 is located at 0xFFFF.F900 + 0x20, etc.

Field #	Address	Field Name
3	(base) + 0xc	SP for the kernel event handler
2	(base) + 0x8	kernel exception handler address
1	(base) + 0x4	SP for the kernel TLB-Refill event handler
0	(base) + 0x0	kernel TLB-Refill event handler address

Table 3: Pass Up Vector Layout

12 Device Registers

All external devices share the same device register structure. While each device class has a specific use and format for these fields, all device classes, except terminal devices, use:

- COMMAND to allow commands to be issued to the device controller.
- STATUS for the device controller to communicate the device status to the processor.
- DATA0 & DATA1 to pass additional parameters to the device controller or the passing of data from the device controller.

Field #	Address	Field Name
3	(base) + 0xc	DATA1
2	(base) + 0x8	DATA1
1	(base) + 0x4	COMMAND
0	(base) + 0x0	STATUS

Table 4: Device Register Layout

All 40 device registers in μ RISC-V are located in low memory starting at 0x1000.0054 as in Figure 1. `/usr/include/uriscv/types.h` contains C-language struct definitions for an individual device register and the collection of devices registers.

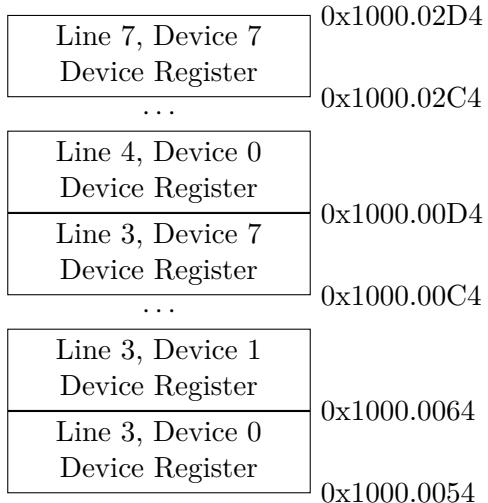


Figure 1: Device Registers Area

12.1 Terminal Devices

μ RISC-V supports up to eight serial terminal device interfaces. Each terminal interface contains two sub-devices; a *transmitter* and a *receiver*. These two sub-devices operate independently and concurrently. To support the two-subdevices a terminal interface's device register is redefined as follows:

Field #	Address	Field Name
3	(base) + 0xc	TRANSM_COMMAND
2	(base) + 0x8	TRANSM_STATUS
1	(base) + 0x4	RECV_COMMAND
0	(base) + 0x0	RECV_STATUS

Table 5: Terminal Device Register Layout

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-block count, blocked PCBs lists/pointers, etc.).
2. `scheduler.c` This module implements the Scheduler.
3. `interrupts.c` This module implements the device/timer interrupt exception handler.
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`).

13.2 Accessing the `liburiscv` Library

Accessing the 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.²

The C `liburiscv` functions to access the registers and BIOS-implemented services are:

- `void WAIT():` Idle Processor
- `void HALT():` Displays the text "System halted" on terminal 0 and puts the processor into an infinite loop
- `void PANIC():` Displays the text "kernel panic" on terminal 0 and puts the processor into an infinite loop
- `unsigned int LDST(state_t *statep):` atomically load the processor state with the state located at the supplied address. `statep` is the address where the processor state is stored.
- `unsigned int LDCXT(unsigned int stackPtr, unsigned int status, unsigned int pc):` LDCXT allows a current process to change its operating mode/context: turn on/off interrupt masks, turn on user mode, and at the same time change the location of execution.

²The file `liburiscv.h` is part of the PandOSsh distribution. `/usr/include/uriscv/` is the recommended installation location for this file.

C Usage	Register
<code>unsigned int getIndex()</code>	<code>Index</code>
<code>unsigned int getENTRYHI()</code>	<code>EntryHi</code>
<code>unsigned int getRandom()</code>	<code>Random</code>
<code>unsigned int getENTRYLO()</code>	<code>EntryLo</code>
<code>unsigned int getBADVADDR()</code>	<code>BadVaddr</code>
<code>unsigned int getStatus()</code>	<code>Status</code>
<code>unsigned int getCause()</code>	<code>Cause</code>
<code>unsigned int getMIE()</code>	<code>Mie</code>
<code>unsigned int getMIP()</code>	<code>Mip</code>
<code>unsigned int getEPC()</code>	<code>EPC</code>
<code>unsigned int getTIMER()</code>	<code>Timer</code>

Table 6: Register Read Commands

C Usage	Register
<code>unsigned int setIndex()</code>	<code>Index</code>
<code>unsigned int setENTRYHI()</code>	<code>EntryHi</code>
<code>unsigned int setRANDOM()</code>	<code>Random</code>
<code>unsigned int setENTRYLO()</code>	<code>EntryLo</code>
<code>unsigned int setBADVADDR()</code>	<code>BadVaddr</code>
<code>unsigned int setStatus()</code>	<code>Status</code>
<code>unsigned int setCause()</code>	<code>Cause</code>
<code>unsigned int setMIE()</code>	<code>Mie</code>
<code>unsigned int setMIP()</code>	<code>Mip</code>
<code>unsigned int setEPC()</code>	<code>EPC</code>
<code>unsigned int setTIMER()</code>	<code>Timer</code>

Table 7: Register Write Commands

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 `cmake` program to maintain your code.

Once your source files (from Phase 1 and Phase 2) 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 generated `Makefile` by `cmake`), your code can be tested by launching the μ RISC-V emulator.

The test program reports on its progress by writing messages to terminal 0. At the conclusion of the test program, either successful or unsuccessful, μ RISC-V 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!