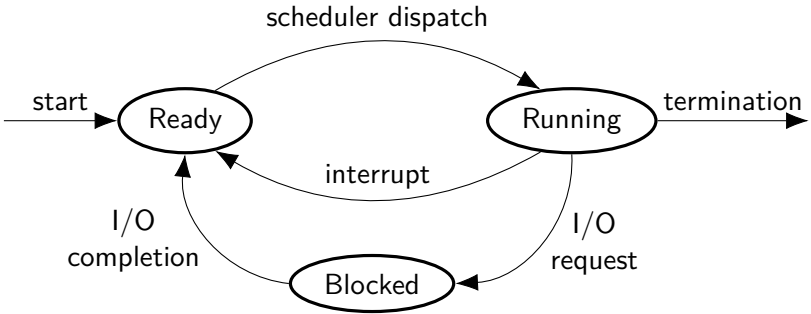
MultiPandOS: Phase 2

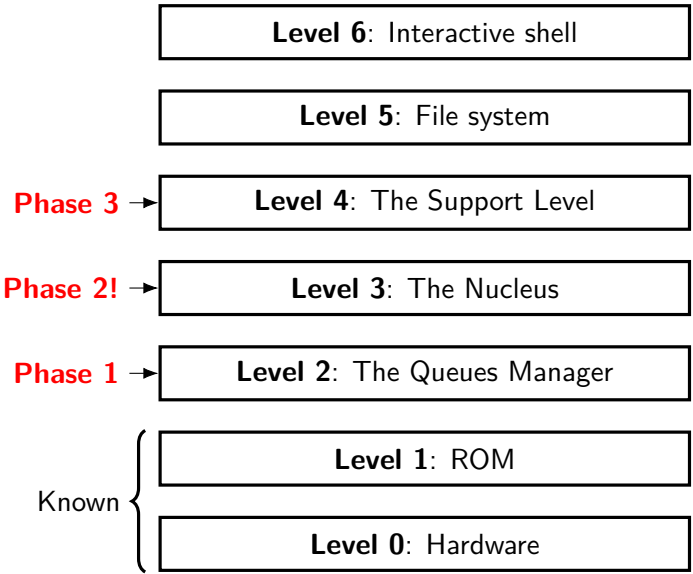
Luca Bassi (luca.bassi14@studio.unibo.it)
Luca Orlandello (luca.orlandello@studio.unibo.it)

December 18, 2024

Process life cycle



MultiPandOS: six levels of abstraction



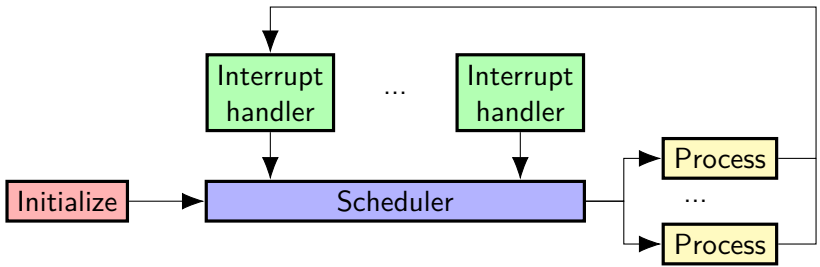
MultiPandOS: Phase 2 - Level 3: The Nucleus

MultiPandOS is a multiprocessor kernel.

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

- Nucleus initialization
- The Scheduler
- Exception handling and SYSCALL processing
- Device interrupt handler
- The passing up of the handling of all other events

Kernel schema



MultiPandOS: Phase 2 - Level 3: The Nucleus

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

- Receives the control flow from the exception handling facility of Level 1 (the ROM): TLB-Refill events and all other exception types (interrupts, system calls, TLB exceptions, program traps)
- 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 and multiprocessor

MultiPandOS: Phase 2 - Level 3: The Nucleus

In summary, after some one-time Nucleus initialization code, the Nucleus will repeatedly dispatch a process.

This Current Process will run until:

- It makes a system call
- It terminates
- The timer assigned to the Scheduler generates an interrupt
- A device interrupt occurs

MultiPandOS: Phase 2 - Level 3: The Nucleus

If the Scheduler ever discovers that the Ready Queue is empty it will:

- 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

Nucleus Initialization

The entry point for MultiPandOS (i.e. `main()`) performs the Nucleus initialization, which includes:

- Declare the Level 3 global variables: Process Count, Ready Queue, Current Process, Blocked PCBs, Device semaphores, Global Lock
- Populate the Processors Pass Up Vector: TLB-Refill event handler, exception handler
- Initialize the Level 2 (Phase 1) data structures
- Initialize all the previously declared variables
- Load the system-wide Interval Timer with 100 milliseconds
- Instantiate the test process
- Interrupt Routing
- Set the state for the other `NCPU - 1` CPUs
- Call the Scheduler

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.

In its simplest form whenever the Scheduler is called it should dispatch the "next" process in the Ready Queue.

- Remove the PCB from the head of the Ready Queue and store the pointer to the PCB in the Current Process field of the current CPU
- Load 5 milliseconds on the Processor's Local Timer
- Perform a Load Processor State on the processor state stored in PCB of the Current Process

Dispatching a process transitions it from a "ready" process to a "running" process.

The Scheduler

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

- If the Process Count is 0, invoke the `HALT` BIOS service/instruction
- If the Process Count > 0 enter a Wait State

TLB-Refill events

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.

The code for this function, for Level 3/Phase 2 testing purposes is in the specs.

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

Exception Handling

At startup, the Nucleus will have populated the CPUs' Pass Up Vector with the address of the Nucleus exception handler 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 (0x0FFF.F000, constant `BIOSDATAPAGE`).

You can use the `GET_EXCEPTION_STATE_PTR(id)` macro to access the BIOS Data Page of the various CPUs.

Exception Handling

The cause of this exception is encoded in the .ExcCode field of the Cause register (Cause.ExcCode) in the saved exception state.

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

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.

SYSCALL Exception Handling: 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 the initial state for the newly created process. The process requesting the NSYS1 service continues to exist and to execute.

```
int retValue = SYSCALL(CREATEPROCESS, state t *statep,
    0, 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.

SYSCALL Exception Handling: 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.

```
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.

SYSCALL Exception Handling: Passeren (P) (NSYS3)

This service requests the Nucleus to perform a P operation on a binary 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, control is either returned to the Current Process of the current processor, or this process is blocked on the ASL (transitions from "running" to "blocked") and the Scheduler is called.

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

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

SYSCALL Exception Handling: Verhogen (V) (NSYS4)

This service requests the Nucleus to perform a V operation on a binary 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. Depending on the value of the semaphore, control is either returned to the Current Process of the current processor, or this process is blocked on the ASL (transitions from "running" to "blocked") and the Scheduler is called.

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

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

SYSCALL Exception Handling: DoIO (NSYS5)

MultiPandOS 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. 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.

SYSCALL Exception Handling: DoIO (NSYS5)

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.

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

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

Important: Write the command to the device register as last operation before calling the scheduler otherwise race conditions can happen.

SYSCALL Exception Handling: 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.

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

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

SYSCALL Exception Handling: 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.

Since the Pseudo-clock semaphore is a synchronization semaphore, this call should always block the Current Process of the current CPU 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.

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

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

SYSCALL Exception Handling: 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.

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

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

SYSCALL Exception Handling: GetProcessID (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.

NSYS1-NSYS9 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.

Returning from a SYSCALL Exception

For SYSCALLs calls that do not block or terminate, control is returned to the Current Process of the current processor at the conclusion of the Nucleus's SYSCALL exception handler.

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 prior to returning control to the interrupted execution stream.

Blocking SYSCALLs

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

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

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 Pass Up Vector was properly initialized by the Nucleus as part of Nucleus initialization, and that the Nucleus exception handler correctly decodes that the current exception is an interrupt using `CAUSE_IS_INT`, control should be passed to one's Nucleus interrupt exception handler.

Depending on the interrupt exception code we update a local variable (more information on the specifications): 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.

Interrupt Exception Handling

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.

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.

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

Non-Timer Interrupts

1. Calculate the address for this device's device register
2. Save off the status code from the device's device register
3. Acknowledge the outstanding interrupt
4. Perform a V operation on the Nucleus maintained semaphore associated with this (sub)device
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 of the current CPU if exists otherwise call the scheduler

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 whenever it dispatches a process.

This "running" process will either:

- Terminate
- 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.

Processor Local Timer (PLT) Interrupts

The PLT portion of the interrupt exception handler should therefore:

- Acknowledge the PLT interrupt by loading the timer with a new value
- 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

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. 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.

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 System-wide Interval Timer and the Pseudo-clock

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
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

Pass Up Vector and 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.

- 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".

SYSCALL Exceptions Numbered by positive numbers

A SYSCALL exception numbered 1 and above occurs when the Current Process 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.

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.

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

TLB Exception Handling

A TLB exception occurs when μRISC-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.

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

Accumulated CPU Time

μRISC-V 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.

Accumulated CPU Time

A field has been defined in the PCB for this purpose (`p_time`). Hence GetCPUTime 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.

Accumulated CPU Time

The three timer devices are mechanisms for implementing MultiPandOS'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?
- 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?

Process Termination

When a process is terminated 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
- The process count variable need to be adjusted accordingly
- Processes can't hide: PCB is either the Current Process ("running"), sitting on the Ready Queue ("ready"), blocked waiting for device or for non-device ("blocked")

Testing

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

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 μRISC-V 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, μRISC-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.

Submission

The deadline is set for **Tuesday April 22, 2024 at 23:59** or **Sunday May 18, 2024 at 23:59**.

Upload a single `phase2.tar.gz` in the folder associated to your group with:

- All the source code with a Makefile
- Documentation
- README and AUTHOR files

Please comment your code!

We will send you an email with the hash of the archive, you should check that everything is correct.

You will receive another email with the score out of 10.