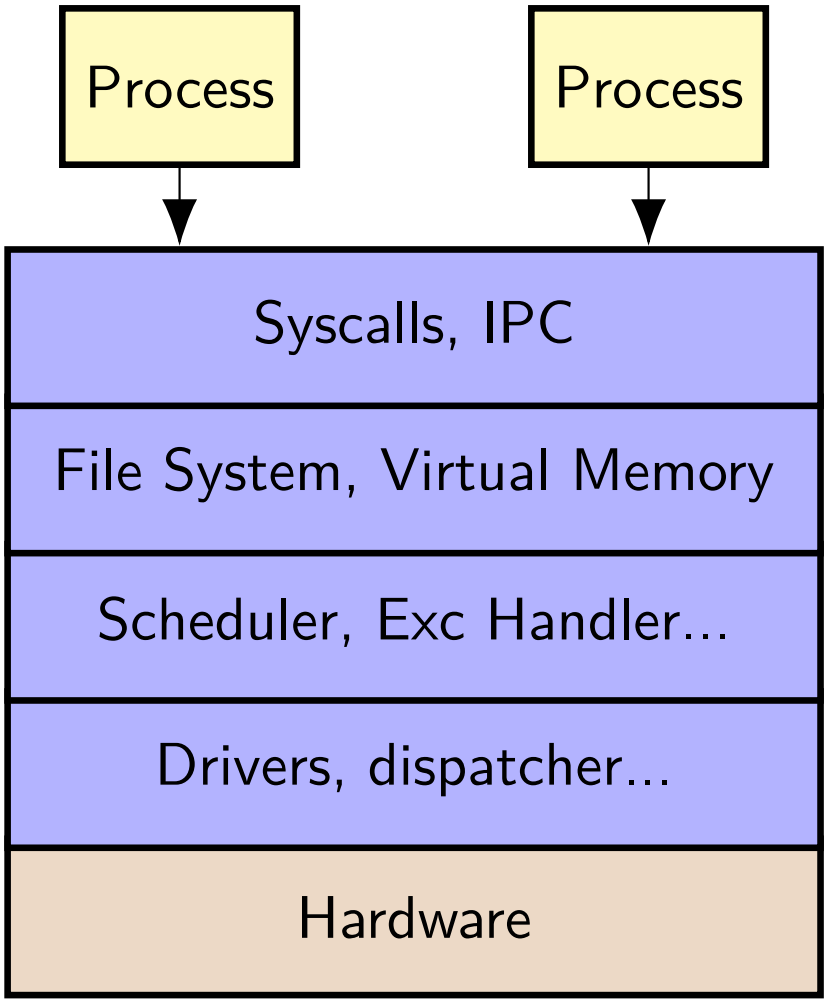
Kernel and μPandOS: Phase 2

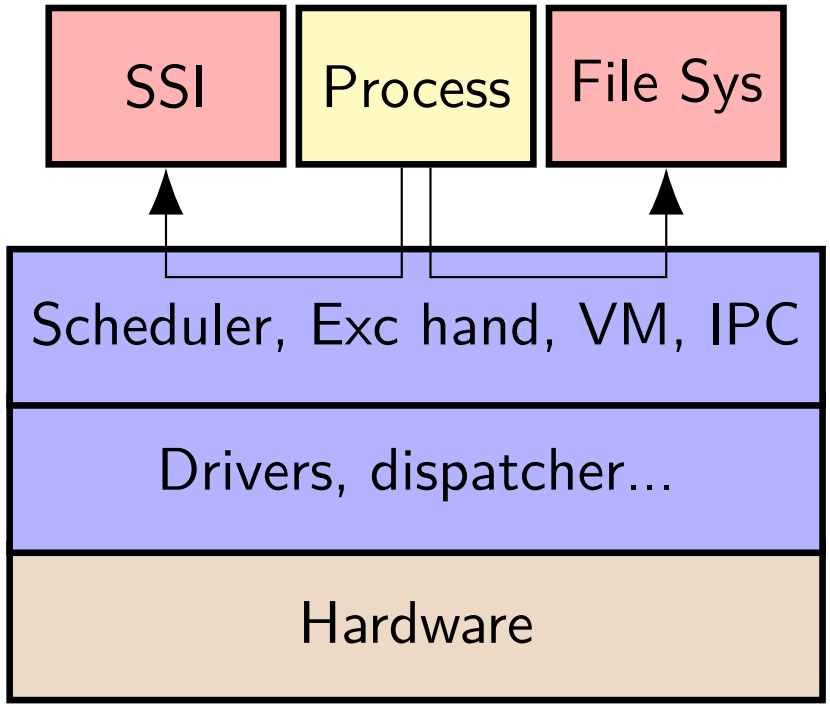
Luca Bassi (luca.bassi14@studio.unibo.it)
Gabriele Genovese (gabriele.genovese2@studio.unibo.it)

December 6, 2023

Monolithic Kernel vs Microkernel

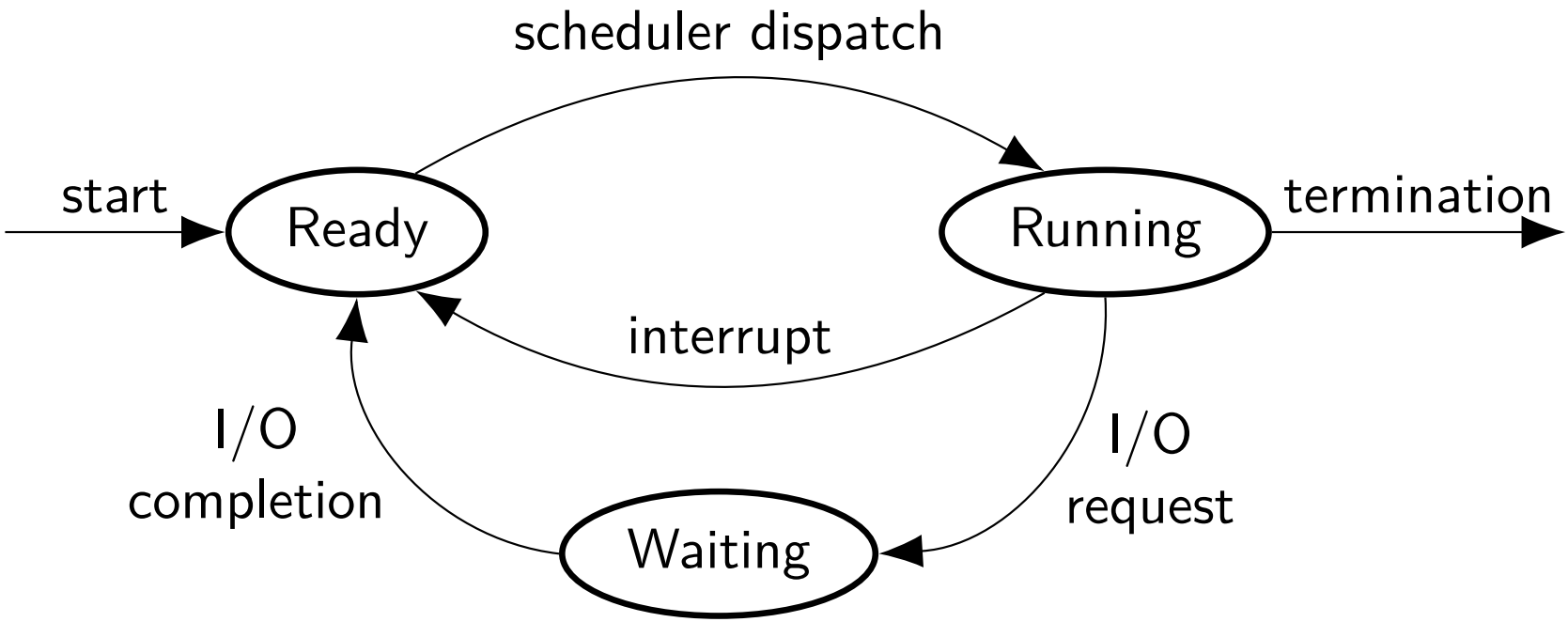


Monolithic kernel

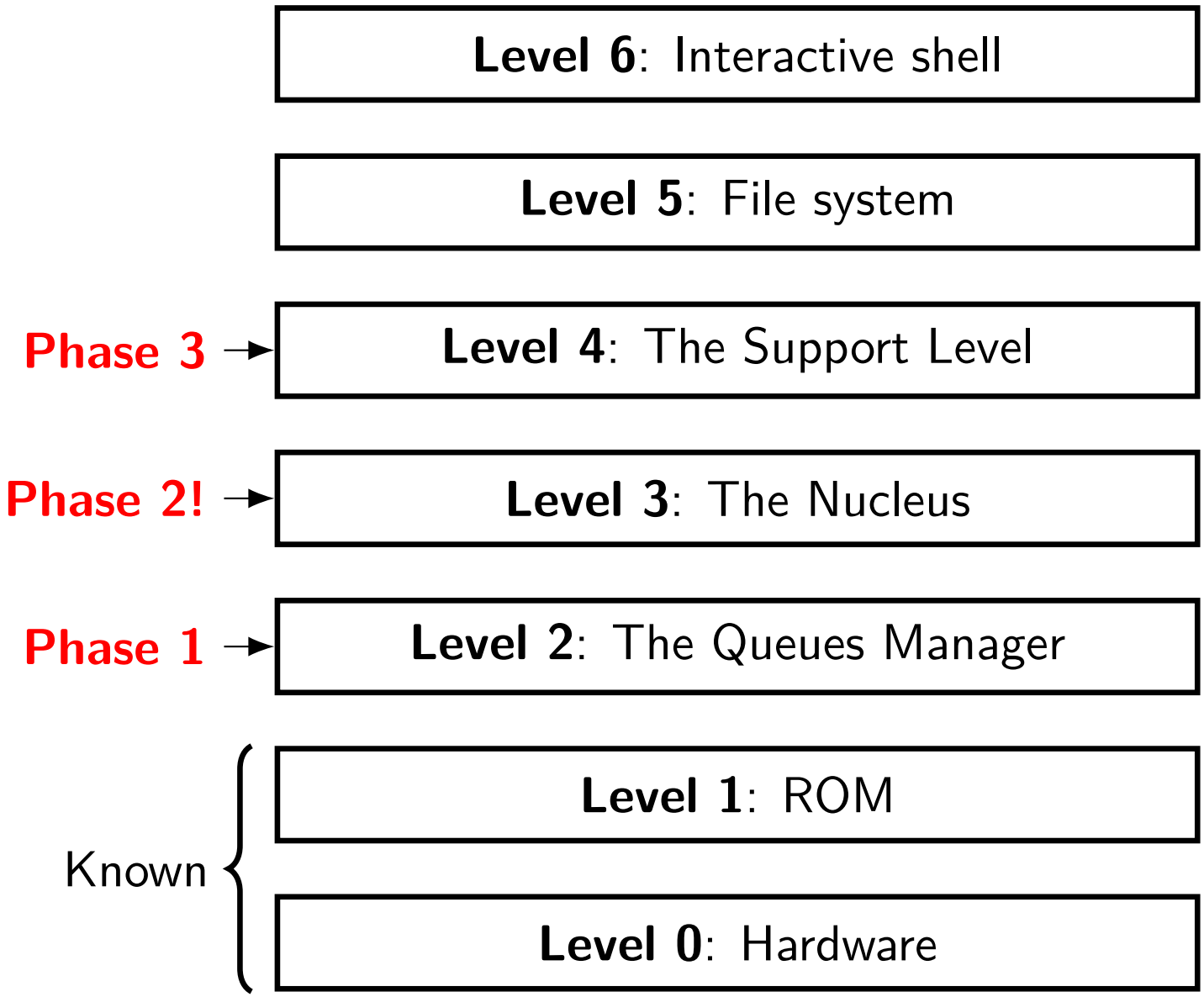


Microkernel

Process life cycle



μPandOS: six levels of abstraction



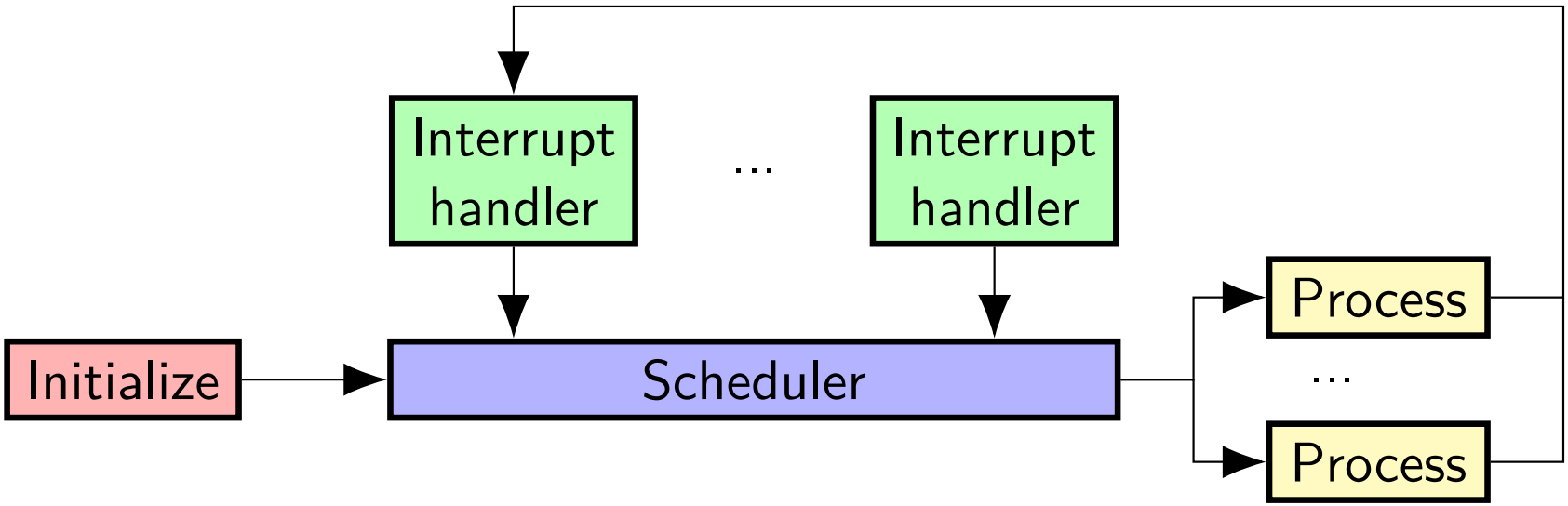
μPandOS: Phase 2 - Level 3: The Nucleus

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

Kernel schema



μPandOS: 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.

Important: In this phase, there will be some differences between μMPS3 and μRISCV, check the specs for the details.

μPandOS: 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

μPandOS: 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
- PANIC: halt execution in the presence of deadlock

Nucleus Initialization

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

- Declare the Level 3 global variables: Process Count, Soft-block Count, Ready Queue, Current Process, Blocked PCBs
- Populate the Processor 0 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 SSI process
- Instantiate the test process
- 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
- 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 1 and the SSI is the only process in the system, invoke the `HALT` BIOS service/instruction
- If the Process Count > 0 and the Soft-block Count > 0 enter a Wait State
- If the Process Count > 0 and the Soft-block Count is 0 invoke the `PANIC` BIOS service/instruction

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 Processor 0 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 will have been stored at the start of the BIOS Data Page.

Exception Handling

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

- For exception code 0 (Interrupts), processing should be passed along to your Nucleus's device interrupt handler
- For exception codes 1-3 (TLB exceptions), processing should be passed along to your Nucleus's TLB exception handler
- For exception codes 4-7, 9-12 (Program Traps), processing should be passed along to your Nucleus's Program Trap exception handler
- For exception code 8 (SYSCALL), processing should be passed along to your Nucleus's SYSCALL 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: 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`). This system call doesn't require the process to lose its remaining time slice.

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.

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

SYSCALL Exception Handling: 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 may cause the process to lose its remaining time slice, since if its inbox is empty it has to be frozen.

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.

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

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.

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.

In any event the PC that was saved is 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 (SYS2), a number of steps need to be performed:

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

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 μPandOS, 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.

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.

System Service Interface

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;
}
```

A process makes a service request by using a specific function:

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

System Service Interface

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
- 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

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.

Nucleus Services: 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.

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.

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.

Nucleus Services: TerminateProcess

This services causes the sender 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.

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.

Nucleus Services: DoIO

μPandOS 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.

Nucleus Services: DoIO

```
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.

Nucleus Services: GetCPUTime

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

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

Nucleus Services: 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. 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.

Nucleus Services: 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.

Nucleus Services: 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.

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 Cause.ExcCode, control should be passed to one's Nucleus interrupt exception handler.

Which interrupt lines have pending interrupts is set in Cause.IP. 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.

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. Send a message and unblock the PCB waiting the status response from this (sub)device
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

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 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.

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 at the time of the exception 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

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), the Current Process will request a WaitForClock service to the SSI.

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 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.

- 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 Cause.ExcCodes of 4-7, 9-12.

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 μMPS3 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.

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

Accumulated CPU Time

μMPS3 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 μPandOS'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
- 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 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 μMPS3 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, μMPS3 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 2, 2024 at 23:59** or **Sunday June 9, 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.