2 Phase 2 - Level 3: The Nucleus

Level 3 (the nucleus) of JaeOS builds on previous levels in two key ways:

- Building on the exception handling facility of Level 1 (the ROM-Excpt handler), provide the exception handlers that the ROM-Excpt handler passes exception handling up to. There will be one exception handler for each type of exception: Program Traps (PgmTrap), SYSCALL/ Breakpoint (SYS/Bp), TLB Management (TLB), and Interrupts (Ints).
- Using the data structures from Level 2 (Phase 1), and the facility to handle both SYS/Bp exceptions and Interrupts - timer interrupts in particular - provide a process scheduler.

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 PgmTrap, TLB exceptions and certain SYS/Bp exceptions to the next level; the VM-I/O support level Level (Phase 3). Since virtual memory is not supported by JaeOS until Level 4, all addresses at this level are assumed to be physical addresses. Nevertheless, the nucleus needs to preserve the state of each process. e.g. If a process is executing with virtual memory on (`CP15_Control[0]=1`) when it is either interrupted or executes a `SYSCALL`, then `CP15_Control` should still be set to 1 when it continues its execution.

2.1 The Scheduler

Your nucleus should guarantee finite progress; consequently, every ready process will eventually have an opportunity to execute. Processes in JaeOS belong to four different priority levels: `PRIO_LOW`, `PRIO_NORM`, `PRIO_HIGH` and `PRIO_IDLE`. The latter is reserved for a special Idle Process that "twiddles its thumbs" by endlessly putting the processor in a Wait State, waiting for an interrupt to occur. μARM supports a `WAIT` ROM service expressly for this purpose.

The scheduler uses a 5 millisecond time slice selecting the running process among the highest priority ready processes. The scheduler also needs to perform some simple deadlock detection and if deadlock is detected perform some appropriate action; e.g. invoke the `PANIC` ROM service/instruction. We define the following

- Ready Queues: Four queues of ProcBlk's (`struct list_head`) representing processes that are ready and waiting for a turn at execution (one queue per priority level).
- Current Process: A pointer to a ProcBlk that represents the current executing process.
- Process Count: The count of the number of processes in the system.
- Soft-block Count: The number of processes in the system currently blocked and waiting for an interrupt; either an I/O to complete, or a timer to expire.

The scheduler should behave in the following manner if the three main Ready Queues are empty (i.e. the Idle Process has to be executed):

1. If the Process Count is zero invoke the `HALT` ROM service/instruction.
2. Deadlock for JaeOS is defined as when the Process Count > 0 and the Soft-block Count is zero. Take an appropriate deadlock detected action. (e.g. Invoke the `PANIC` ROM service/instruction.)
3. Continue execution loading the Idle Process.

2.2 Nucleus Initialization

Every program needs an entry point (i.e. `main()`), even JaeOS. The entry point for JaeOS performs the nucleus initialization, which includes:

1. Populate the four New Areas in the ROM Reserved Frame. For each New processor state:
    - Set the `PC` to the address of your nucleus function that is to handle exceptions of that type.
    - Set the `SP` to RAMTOP. Each exception handler will use the last frame of RAM for its stack.
    - Set the `CPSR` register to mask all interrupts and be in kernel-mode (System mode).
2. Initialize the Level 2 (Phase 1 - see Chapter 2) data structures:

    ```
    initPcbs()
    initASL()
    ```

3. Initialize all nucleus maintained variables: Process Count, Soft-block Count, Ready Queues, and Current Process.
4. Initialize all nucleus maintained semaphores. In addition to the above nucleus variables, there is one semaphore variable for each external (sub)device in μARM, plus a semaphore to represent a pseudo-clock timer. Since terminal devices are actually two independent sub-devices (see Section 5.7-pops), the nucleus maintains two semaphores for each terminal device. All of these semaphores need to be initialized to zero.
5. Instantiate a single process and place its ProcBlk in the `PRIO_NORM` Ready Queue. A process is instantiated by allocating a ProcBlk (i.e. `allocPcb()`), and initializing the processor state that is part of the ProcBlk. In particular this process needs to have interrupts enabled, virtual memory off, the processor Local Timer enabled, kernel-mode on, `SP` set to RAMTOP-FRAMESIZE (i.e. use the penultimate RAM frame for its stack), and its `PC` set to the address of `test`. 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

   ```
   . . . = (memaddr)test
   ```

   where `memaddr`, in `const.h` has been aliased to `unsigned int`. Remember to declare the test function as external in your program by including the line:

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

6. Set-up the Idle Process and add it to its proper Ready Queue.
7. Call the scheduler.

Once main() calls the scheduler its task is completed since control will 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 the device interrupts and exceptions when they occur. At boot/reset time the nucleus is loaded into RAM beginning with the second frame of RAM; 0x0000.8000. The first frame of RAM is the ROM Reserved Frame. Furthermore, the processor will be in kernel-mode with virtual memory disabled and all interrupts masked. The `PC` is assigned 0x0000.8000 and the `SP`, which was initially set to RAMTOP at boot-time, will now be some value less than RAMTOP due to the activation record for `main()` that now sits on the stack.

2.3 SYS/Bp Exception Handling

A SYSCALL or Breakpoint exception occurs when a SYSCALL or BREAK assembler instruction is executed. Assuming that the SYS/Bp New Area in the ROM Reserved Frame was correctly initialized during nucleus initialization, then after the processor's and ROM-Excpt handler's actions when a SYSCALL or Breakpoint exception is raised, execution continues with the nucleus's SYS/Bp exception handler. A SYSCALL exception is distinguished from a Breakpoint exception by the value of the SWI instruction which raised the exception. SYSCALL exceptions are recognized via an exception code of Sys (8) while Breakpoint exceptions are recognized via an exception code of Bp (9). By convention the executing process places appropriate values in user registers a1-a4 immediately prior to executing a SYSCALL or BREAK instruction. The nucleus will then perform some service on behalf of the process executing the SYSCALL or BREAK instruction depending on the value found in a1. In particular, if the process making a SYSCALL request was in kernel-mode and a1 contained a value in the range [1..10] then the nucleus should perform one of the services described below.

2.3.1 Create Process (SYS1)

When requested, this service causes a new process, said to be a progeny of the caller, to be created. a2 should contain the physical address of a processor state area at the time this instruction is executed and a3 should contain the priority level (PRIO_LOW, PRIO_NORM, PRIO_HIGH). This processor state should be used as the initial state for the newly created process. The process requesting the SYS1 service continues to exist and to execute. If the new process cannot be created due to lack of resources (for example no more free ProcBlk's) or the priority level is not an acceptable value (including PRIO_IDLE), an error code of -1 is placed/returned in the caller's a1, otherwise, return the PID of the newly created process in a1. The SYS1 service is requested by the calling process by placing the value 1 in a1, the physical address of a processor state in a2, the priority level in a3, and then executing a SYSCALL instruction. The following C code can be used to request a SYS1:

```
int SYSCALL (CREATEPROCESS, state t *statep, priority_enum prio)
```

Where the mnemonic constant CREATEPROCESS has the value of 1.

2.3.2 Terminate Process (SYS2)

This services causes the executing process or one process in its progeny to cease to exist. In addition, recursively, all progeny of the designated process are terminated as well. Execution of this instruction does not complete until all progeny are terminated. The SYS2 service is requested by the calling process by placing the value 2 in `a1`, and the value of the designated process' PID in `a2` and then executing a `SYSCALL` instruction. When `a2` is set to zero, SYS2 terminates the calling process. The following C code can be used to request a SYS2:

```
void SYSCALL (TERMINATEPROCESS, pid_t pid)
```

Where the mnemonic constant TERMINATEPROCESS has the value of 2.

2.3.3 Verhogen (V) (SYS3)

When this service is requested, it is interpreted by the nucleus as a request to perform a weighted V operation on a semaphore. The V or SYS3 service is requested by the calling process by placing the value 3 in `a1`, the physical address of the semaphore to be V'ed in `a2`, the weight value (number of resources to be freed) in `a3`, and then executing a `SYSCALL` instruction. The following C code can be used to request a SYS3:

```
void SYSCALL (VERHOGEN, int *semaddr, int weight)
```

Where the mnemonic constant VERHOGEN has the value of 3.

2.3.4 Passeren (P) (SYS4)

When this services is requested, it is interpreted by the nucleus as a request to perform a weighted P operation on a semaphore. The P or SYS4 service is requested by the calling process by placing the value 4 in `a1`, the physical address of the semaphore to be P'ed in `a2`, the weight value (number of resources to be allocated) in `a3`, and then executing a `SYSCALL` instruction. The following C code can be used to request a SYS4:

```
void SYSCALL (PASSEREN, int *semaddr, int weight)
```

Where the mnemonic constant PASSEREN has the value of 4.

2.3.5 Specify Exception State Vector (SYS5)

When this service is requested, the kernel stores the pointers to six processor state areas for exception pass-up. The SYS5 service is requested by the calling process by placing the value 5 in `a1`, and an array of six processor state pointers in `a2`. These pointers are the addresses of the OLD area for TLB exceptions, the NEW area for TLB exceptions, the OLD and NEW areas for SYSCALL and the OLD and NEW area for Program Trap, respectively. When an exception occurs for which an Exception State Vector has been specified for, the nucleus stores the processor state at the time of the exception in the area pointed to by the address in the specific OLD area, and loads the new processor state from the area pointed to by the address given in the corresponding NEW area. A process may request SYS5 service at most once. An attempt to request a SYS5 service more than once by any process should be construed as an error and treated as a SYS2 of the process itself. If an exception occurs while running a process which has not specified an Exception State Vector for that exception type, then the nucleus should treat the exception as a SYS2 as well. The following C code can be used to request a SYS5:

```
void SYSCALL (SPECTRAPVEC, state t **state_vector)
```

Where the mnemonic constant SPECTRAPVEC has the value of 5.

2.3.6 Get CPU Time (SYS6)

When this service is requested, it causes the processor time (in microseconds) used by the requesting process, both as global time and user time, to be returned to the calling process (global time is the time spent by the processor for the calling process, including kernel time and user time). This means that the nucleus must record (in the ProcBlk) the amount of processor time used by each process. The SYS6 service is requested by the calling process by placing the value 6 in `a1`, the address of a `cputime_t` variable in `a2`, the address of a second `cputime_t` variable in `a3` and then executing a `SYSCALL` instruction. At SYS6 completion, the global processor time should be stored at the address specified as `a2` while the processor time in user time should be stored at the address specified as `a3`. The following C code can be used to request a SYS6:

```
void SYSCALL (GETCPUTIME, cputime_t *global, cputime_t *user)
```

Where the mnemonic constant GETCPUTIME has the value of 6.

2.3.7 Wait For Clock (SYS7)

This instruction performs a P operation on the nucleus maintained pseudo-clock timer semaphore. This semaphore is V'ed every 100 milliseconds automatically by the nucleus. The SYS7 service is requested by the calling process by placing the value 7 in `a1` and then executing a `SYSCALL` instruction. The following C code can be used to request a SYS7:

```
void SYSCALL (WAITCLOCK)
```

Where the mnemonic constant WAITCLOCK has the value of 7.

2.3.8 Wait for IO Device (SYS8)

This service performs a P operation on the semaphore that the nucleus maintains for the I/O device indicated by the values in `a2`, `a3`, and optionally `a4`. (P and V operation on I/O semaphores have always weight equal to 1). Note that terminal devices are two independent sub-devices, and are handled by the SYS8 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 in Section 3.6 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 `a1`. For character transmission and receipt, the status word, in addition to containing a device completion code, will also contain the character transmitted or received. As described below in Section 3.6 it is possible that the interrupt can occur prior to the request for the SYS8 service. In this case the requesting process will not block as a result of the P operation and the interrupting device's status word, which was stored off, can now be placed in `a1` prior to resuming execution. The SYS8 service is requested by the calling process by placing the value 8 in `a1`, the interrupt line number in `a2`, the device number in `a3` ([0...7]) , TRUE or FALSE in `a4` to indicate if waiting for a terminal read operation, and then executing a `SYSCALL` instruction. The following C code can be used to request a SYS8:

```
unsigned int SYSCALL (WAITIO, int intlNo, int dnum,
int waitForTermRead)
```

Where the mnemonic constant WAITIO has the value of 8.

2.3.9 Get Process ID (SYS9)

This service returns the Process ID of the caller by placing its value in `a1`. The SYS9 service is requested by the calling process by placing the value 9, in `a1`. The following C code can be used to request a SYS9:

```
pid_t SYSCALL(GETPID)
```

Where the mnemonic constant GETPID has the value of 9.

2.3.10 Get Process Parent ID (SYS10)

This service returns the Process ID of the caller's parent by placing its value in `a1`. The SYS10 service is requested by the calling process by placing the value 10, in `a1`. The following C code can be used to request a SYS10:

```
pid_t SYSCALL(GETPPID)
```

Where the mnemonic constant GETPPID has the value of 10.

2.3.11 SYS1-SYS10 in User-Mode

The above ten 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 PgmTrap exception response. In particular JaeOS should simulate a PgmTrap exception when a privileged service is requested in user-mode. This is done by moving the processor state from the SYS/Bp Old Area to the PgmTrap Old Area, setting `CP15_Cause.ExcCode` in the PgmTrap Old Area to `EXC_RESERVEDINSTR` (Reserved Instruction), and calling JaeOS's PgmTrap exception handler.

2.3.12 Breakpoint Exceptions and SYS11 and Above Exceptions

The nucleus will directly handle all SYS1-SYS10 requests. The ROM-Excpt handler will also directly handle some Breakpoint exceptions; those where the requesting process was executing in kernel-mode and `a1` contained the code for either `LDST`, `PANIC`, or `HALT`. For all other `SYSCALL` and Breakpoint exceptions the nucleus's SYS/Bp exception handler will take one of two actions depending on whether the offending (i.e. Current) process has performed a SYS5:

- If the offending process has NOT issued a SYS5, then the SYS/Bp exception should be handled like a SYS2: the current process and all its progeny are terminated.

- If the offending process has issued a SYS5, the handling of the SYS/Bp exception is passed up. The processor state is moved from the SYS/Bp Old Area into the processor state area whose address was recorded in the ProcBlk as the SYS/Bp Old Area Address. Finally, the processor state whose address was recorded in the ProcBlk as the SYS/Bp New Area Address is made the current processor state.

2.4 PgmTrap Exception Handling

A PgmTrap exception occurs when the executing process attempts to perform some illegal or undefined action. This includes all of the program trap types and reserved instructions. Assuming that the PgmTrap New Area in the ROM Reserved Frame was correctly initialized during nucleus initialization, then after the processor's and ROM-Excpt handler's actions when a PgmTrap exception is raised, execution continues with the nucleus's PgmTrap exception handler. The cause of the PgmTrap exception will be set in `CP15_Cause.ExcCode` in the PgmTrap Old Area. The nucleus's PgmTrap exception handler will take one of two actions depending on whether the offending (i.e. Current) process has performed a SYS5:

- If the offending process has NOT issued a SYS5, then the PgmTrap exception should be handled like a SYS2: the current process and all its progeny are terminated.

- If the offending process has issued a SYS5, the handling of the PgmTrap is passed up. The processor state is moved from the PgmTrap Old Area into the processor state area whose address was recorded in the ProcBlk as the PgmTrap Old Area Address. Finally, the processor state whose address was recorded in the ProcBlk as the PgmTrap New Area Address is made the current processor state.

2.5 TLB Exception Handling

A TLB exception occurs when μARM fails in an attempt to translate a virtual address into its corresponding physical address. Assuming that the TLB New Area in the ROM Reserved Frame was correctly initialized during nucleus initialization, then after the processor's and ROM-Excpt handler's actions when a TLB exception is raised, execution continues with the nucleus's TLB exception handler. The cause of the TLB exception will be set in `CP15_Cause.ExcCode` in the TLB Old Area. The nucleus's TLB exception handler will take one of two actions depending on whether the offending (i.e. Current) process has performed a SYS5:

- If the offending process has NOT issued a SYS5, then the TLB exception should be handled like a SYS2: the current process and all its progeny are terminated.

- If the offending process has issued a SYS5, the handling of the PgmTrap is passed up. The processor state is moved from the TLB Old Area into the processor state area whose address was recorded in the ProcBlk as the TLB Old Area Address. Finally, the processor state whose address was recorded in the ProcBlk as the TLB New Area Address is made the current processor state.

2.6 Interrupt Exception Handling

A device interrupt occurs when either a previously initiated I/O request completes or when the Interval Timer makes a `0x0000.0000` $\rightarrow$ `0xFFFF.FFFF` transition. Assuming that the Ints New Area in the ROM Reserved Frame was correctly initialized during nucleus initialization, then after the processor's and ROM-Excpt handler's actions when an Ints exception is raised, execution continues with the nucleus's Ints exception handler. Which interrupt lines have pending interrupts is set in `CP15_Cause.IP`. Furthermore, for interrupt lines 3–7 the Interrupting Devices Bit Map, as defined in the μARM informal specifications document, will indicate which devices on each of these interrupt lines have a pending interrupt. It is important to note that 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 pending interrupts simultaneously as well. You are strongly encouraged to 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 only processes 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. As described in Section 5.7-pops, terminal devices are actually two sub-devices; a transmitter and a receiver. These two sub-devices operate independently and concurrently. 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 processor Interval Timer (interrupt line 2) is the highest priority interrupt and reading from terminal 7 (interrupt line 7, device 7; read) is the lowest priority interrupt. The nucleus's Interrupts exception handler will perform a number of tasks:

- Acknowledge the outstanding interrupt. For all devices except the Interval Timer this is accomplished by writing the acknowledge command code in the interrupting device's `COMMAND` device register. Alternatively, writing a new command in the interrupting device's device register will also acknowledge the interrupt. An interrupt for a timer device is acknowledged by loading the timer with a new value.

- Perform a V operation (weight 1) on the nucleus maintained semaphore associated with the interrupting (sub)device if the semaphore has value less than 1. The nucleus maintains two semaphores for each terminal sub-device. For Interval Timer interrupts that represent a pseudo-clock tick (see Section 3.7.1), perform the V operation on the nucleus maintained pseudo-clock timer semaphore. A V operation on the pseudo clock should unlock all the waiting processes.

- If the SYS8 for this interrupt was requested prior to the handling of this interrupt, recognized by the V operation above unblocking a blocked process, store the interrupting (sub)device's status word in the newly unblocked process's `a1`. If the SYS8 for this interrupt has not yet been requested, recognized by the V operation not unblocking any process, store off the interrupting device's status word until the SYS8 is eventually requested.

2.7 Nuts and Bolts

2.7.1 Timing Issues

While μARM has two clocks, the TOD clock and Interval Timer, only the Interval Timer can generate interrupts. Hence the Interval Timer must be used simultaneously for two purposes: generating interrupts to signal the end of processes' time slices, and to signal the end of each 100 millisecond period (a pseudo-clock tick); i.e. the time to V the semaphore associated with the pseudo-clock timer. It is insufficient to simply V the pseudo-clock timer's semaphore after every 20 time slices; processes may block (SYS4, SYS7, or SYS8) or terminate (SYS2) long before the end of their current time slice. A more careful accounting method is called for; one where some (most) of the Interval Timer interrupts represent the conclusion of a time slice while others represent the conclusion of a pseudo-clock tick. Furthermore, when no process requests a SYS7, the pseudo-clock timer semaphore, if left unadjusted, will grow by 1 every 100 milliseconds. This means that if a process, after 500 milliseconds requests a SYS7, and there were no intervening SYS7 requests, it will not block until the next pseudo-clock tick as hoped for, but will immediately resume its execution stream. Therefore at each pseudo-clock tick, if no process was unblocked by the V operation (i.e. the semaphore's value after the increment performed during the V operation was greater than zero), the semaphore's value should be decremented by one (i.e.

reset to zero). The opposite is also true; if more than one process requests a SYS7 in between two adjacent pseudo-clock ticks then at the next pseudo-clock tick, all of the waiting processes should be unblocked and not just the process that was waiting the longest. The CPU time used by each process must also be kept track of (i.e. SYS6). This implies an additional field to be added to the ProcBlk structure. While the Interval Timer is useful for generating interrupts, the TOD clock is useful for recording the length of an interval. 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. The Interval Timer and TOD clock are mechanisms for implementing JaeOS's policy. 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 purposes, which process, if any, should be charged with this time.

2.7.2 Returning from a SYS/Bp Exception

`SYSCALL`'s that do not result in process termination return control to the requesting process's execution stream. This is done either immediately (e.g. SYS6) or after the process is blocked and eventually unblocked (e.g. SYS8). In any event the `PC` that was saved is, for this kind of exceptions, the address of the address of the `SYSCALL` assembly instruction that caused the exception.

2.7.3 Loading a New Processor State

It is the job of the ROM-Excpt handler to load new processor states; either as part of passing up exception handling (the loading of the processor state from the appropriate New Area) or for `LDST` processing. Whenever the ROM-Excpt handler loads a processor state, the previous state of the running process get stored in the corresponding OLD area. Any information concerning the running process (prior to the exception) should be retrieved from that OLD state, e.g. the CPRS register fields are useful to state the running execution mode.

2.7.4 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; its parent can no longer have this ProcBlk as one of its progeny.
- If the value of a semaphore is negative, it means that some processes (of which the ProcBlks are blocked on that semaphore) have requested a number of resources. Hence if a terminated process is blocked on a semaphore, the value of the semaphore must be adjusted.
- If a terminated process is blocked on a device semaphore, the semaphore should NOT be adjusted. When the interrupt eventually occurs the semaphore will get V'ed by the interrupt handler.
- The process count and soft-blocked variables need to be adjusted accordingly.
- Processes (i.e. ProcBlk's) can't hide. A ProcBlk is either the current process, sitting on a ready queue, blocked on a device semaphore, or blocked on a non-device semaphore.

2.7.5 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, device semaphores, etc.)
2. `interrupts.c` This module implements the device interrupt exception handler. This module will process all the device interrupts, including Interval Timer interrupts, converting device interrupts into V operations on the appropriate semaphores.
3. `exceptions.c` This module implements the TLB, PgmTrap and SYS/Bp exception handlers.
4. `scheduler.c` This module implements JaeOS's process scheduler and deadlock detector.