3 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.

3.1 The Scheduler

Your nucleus should guarantee finite progress; consequently, every ready process will eventually have an opportunity to execute. For simplicitys sake this chapter describes the implementation of a simple round-robin scheduler with a time slice value of 5 milliseconds.

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 Queue: A pointer to a queue of ProcBlks representing processes that are ready and waiting for a turn at execution.
- 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 Ready Queue is empty:

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 Softblock Count is zero. Take an appropriate deadlock detected action. (e.g. Invoke the `PANIC` ROM service/instruction.)

3. If the Process Count > 0 and the Soft-block Count > 0 enter a *Wait State*. A Wait State is a state where the processor is twiddling its thumbs, or waiting until an interrupt to occur. uARM supports a `WAIT` instruction expressly for this purpose.

3.2 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 Queue, 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 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. 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.

3.3 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..11]` then the nucleus should perform one of the services described below.

3.3.1 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. 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. The newly created process is provided with a unique *Process IDentifier*, see section 3.7.5 for a proposed pid selection algorythm. If the new process cannot be created due to lack of resources (for example no more free ProcBlk's), 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`, and then executing a `SYSCALL` instruction.

The following C code can be used to request a SYS1:

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

Where the mnemonic constant CREATEPROCESS has the value of 1.

3.3.2 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.

3.3.3 Semaphore Operation (SYS3)

When this service is requested, it is interpreted by the nucleus as a request to perform a weighted operation on a semaphore. Performing this operation with positive values have the effect of freeing resources associated with the semaphore, on the other hand, requesting the service with negative values allocates resources. When resources are allocated, if the value of the semaphore was or becomes negative, the requesting process should be blocked in the semaphore's queue. When resources are freed, until the amount of resources required by the first blocked process are less or equal to the newly freed ones, the process should be unlocked and placed in the Ready Queue. An attempt to request a SYS3 service with weight 0 should be construed as an error and treated as a SYS2 of the process itself. The 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/allocated) in `a3`, and then executing a `SYSCALL` instruction.

The following C code can be used to request a SYS3:

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

Where the mnemonic constant SEMOP has the value of 3.

3.3.4 Specify Sys/BP Handler (SYS4)

When this service is requested, the kernel prepares the System Call/Breakpoint New Area of the calling process for exception pass-up. The SYS4 service is requested by the calling process by placing the value 4 in `a1`, the address of the handler function in `a2`, the address of the handler's stack in `a3` and the execution flags in `a4`. The last parameter is an unsigned int oranized as follows:

- `flags[0-4]`: processor execution mode
- `flags[6]`: fast interrupts enabled (interval timer)
- `flags[7]`: regualr interrupts enabled (devices)
- `flags[31]`: virtual memory enabled

When a SYS4 is requested, the nucleus stores the handler address into the New Area `pc` register, the stack address into the New Area `sp` register, the first three flags into the New Area `cpsr` register, sets the New Area virtual memory mode according to the flags' value and the New Area ASID to be the same as the calling process's.

A process may request SYS4 service at most once. An attempt to request a SYS4 service more than once by any process should be construed as an error and treated as a SYS2 of the process itself.

The following C code can be used to request a SYS4:

```
void SYSCALL (SPECSYSHDL, memaddr pc, memaddr sp, unsigned int flags)
```

Where the mnemonic constant SPECSYSHDL has the value of 4.

3.3.5 Specify TLB Handler (SYS5)

When this service is requested, the kernel prepares the TLB Exception New Area of the calling process for exception pass-up. The SYS5 service is requested by the calling process by placing the value 5 in `a1`, the address of the handler function in `a2`, the address of the handler's stack in `a3` and the execution flags in `a4` (see 3.3.4).

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.

The following C code can be used to request a SYS5:

```
void SYSCALL (SPECTLBHDL, memaddr pc, memaddr sp, unsigned int flags)
```

Where the mnemonic constant SPECTLBHDL has the value of 5.

3.3.6 Specify Program Trap Handler (SYS6)

When this service is requested, the kernel prepares the Program Trap New Area of the calling process for exception pass-up. The SYS6 service is requested by the calling process by placing the value 6 in `a1`, the address of the handler function in `a2`, the address of the handler's stack in `a3` and the execution flags in `a4` (see 3.3.4).

A process may request SYS6 service at most once. An attempt to request a SYS6 service more than once by any process should be construed as an error and treated as a SYS2 of the process itself.

The following C code can be used to request a SYS6:

```
void SYSCALL (SPECPGMTHDL, memaddr pc, memaddr sp, unsigned int flags)
```

Where the mnemonic constant SPECPGMTHDL has the value of 6.

3.3.7 Exit From Trap (SYS7)

When this service is requested, it causes the processor status stored in one of the three ProcBlk Old Areas to be loaded in order to return from a higher level handler function. Only the nucleus has knowledge about ProcBlks and their content, so higher level handlers need a way to exit from trap handling and optionally return some values to the caller. The SYS7 service is requested by the calling process by placing the value 7 in `a1`, the type of exception in `a2` and the return value in `a3`. The exception type must be 0 for System Calls/Breakpoints, 1 for TLB Exceptions or 2 for Program Traps (these values are aliased in `const.h`). The optional return value should be copied to the Old State `a1` register.

The following C code can be used to request a SYS7:

```
void SYSCALL (EXITTRAP, unsigned int excptype, unsigned int retval)
```

Where the mnemonic constant EXITTRAP has the value of 7.

3.3.8 Get CPU Time (SYS8)

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 SYS8 service is requested by the calling process by placing the value 8 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 SYS8 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 SYS8:

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

Where the mnemonic constant GETCPUTIME has the value of 8.

3.3.9 Wait For Clock (SYS9)

This instruction locks the requesting process on the nucleus maintained pseudo-clock timer semaphore. Each process waiting on this semapgore is unlocked every 100 milliseconds automatically by the nucleus. The SYS9 service is requested by the calling process by placing the value 9 in `a1` and then executing a `SYSCALL` instruction.

The following C code can be used to request a SYS9:

```
void SYSCALL (WAITCLOCK)
```

Where the mnemonic constant WAITCLOCK has the value of 9.

3.3.10 I/O Device Operation (SYS10)

This service performs the operation indicated by the value in `a2` and then locks the requesting process on the semaphore that the nucleus maintains for the I/O device indicated by the values in `a3`, `a4`. (Operations on I/O semaphores have always weight equal to +/-1). Note that terminal devices are two independent sub-devices, and are handled by the SYS10 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 SYS3 operation on the nucleus maintained semaphore whenever that (sub)device generates an interrupt. Once the process resumes after the occurrence of the 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. The SYS10 service is requested by the calling process by placing the value 10 in `a1`, the device command word in `a2`, the interrupt line number in `a3`, the device number in `a4` ([0...7]) with the most significant bit set to 1 if it is a terminal READ operation and then executing a `SYSCALL` instruction.

The following C code can be used to request a SYS10:

```
unsigned int SYSCALL (IODEVOP, unsinged int command, int intlNo, unsigned int dnum)
```

Where the mnemonic constant IODEVOP has the value of 10.

3.3.11 Get Process ID (SYS11)

This service returns the Process ID of the caller by placing its value in `a1`. The SYS11 service is requested by the calling process by placing the value 11, in `a1`.

The following C code can be used to request a SYS11:

```
pid_t SYSCALL(GETPID)
```

Where the mnemonic constant GETPID has the value of 11.

3.3.12 SYS1-SYS11 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.

3.3.13 Breakpoint Exceptions and SYS12 and Above Exceptions

The nucleus will directly handle all SYS1-SYS11 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 ither `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 SYS4:

- If the offending process has NOT issued a SYS4, 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 SYS4, 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 stored in the ProcBlk as the SYS/Bp Old Area, the four parameter register (`a1-a4`) are copied from SYS/Bp Old Area to the ProcBlk SYS/Bp New Area and the lower byte of SYS/Bp Old Area's `cpsr` register is copied in the higher byte of ProcBlk SYS/Bp New Area's `a1` register. Finally, the processor state stored in the ProcBlk as the SYS/Bp New Area is made the current processor state.

3.4 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 stored in the ProcBlk as the PgmTrap Old Area and `Cause` register is copied from the PgmTrap Old Area into the ProcBlk PgmTrap New Area's `a1` register. Finally, the processor state stored in the ProcBlk as the PgmTrap New Area is made the current processor state.

3.5 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 SYS6:

- If the offending process has NOT issued a SYS6, 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 SYS6, the handling of the PgmTrap is passed up. The processor state is moved from the TLB Old Area into the processor state stored in the ProcBlk as the TLB Old Area and `Cause` register is copied from the TLB Old Area into the ProcBlk TLB New Area's `a1` register. Finally, the processor state stored in the ProcBlk as the TLB New Area is made the current processor state.

3.6 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 Intettupts 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 Interrupt exception is raised, execution continues with the nucleus's Interrupts 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 SYS3 operation with 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 SYS3 operation on the nucleus maintained pseudo-clock timer semaphore. A SYS3 operation on the pseudo clock should unlock all the waiting processes.

- If the SYS10 for this interrupt was requested, recognized by the SYS3 operation above unblocking a blocked process, store the interrupting (sub)device's status word in the newly unblocked process'es `a1`.

3.7 Nuts and Bolts

3.7.1 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 unlock the semaphore associated with the pseudo-clock timer. It is insufficient to simply perform a SYS3 on the pseudo-clock timer's semaphore after every 20 time slices; processes may block (SYS3, SYS9, or SYS10) 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 after each pseudo-clock tick, the semaphore's value should always be zero, meaning that any locked process has been freed, if the semaphore's value becomes positive, one or more processes requesting SYS9 operation will misbehave not locking on the semaphore untill next pseudoclock tick. 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.

3.7.2 Returning from a SYS/Bp Exception

`SYSCALL`'s that do not result in process termination return control to the requesting process'es 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 `SYSCALL` assembly instruction that caused the exception.

3.7.3 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.

3.7.4 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 wich the ProcBlks are blocked on that semaphore) have requestes 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 be managed 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.

3.7.5 Process Identifier selection

When a process is created, an identifier is assigned to the process itself, this identifier is a number that must be unique for each process in the system. The method used to select process identifiers is an important aspect in designing the kernel core, a simple algorithm for pid assignment concatenates the index of the new pcb inside the pcb table to a constantly increasing generation number. A more complex selection algorithm is left up to the OS author.

3.7.6 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 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.