The best way to prepare [to be a programmer] is to write programs, and to study great programs that other people have written. In my case, I went to the garbage cans at the Computer Science Center and fished out listings of their operating system.

Bill Gates

3

Phase 2 - Level 3: The Nucleus

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

1. Receives control from the exception handling facility of Level 1. There are two categories of exceptions [Chapter **??**-*pops*]:

   - TLB-Refill events, a relatively frequent occurrence which is triggered during address translation when no matching entries are found in the TLB. Since address translation will not be introduced until the Support Level, the handling of TLB-Refill events is delayed until then.
   - All other exception types, including device/timer interrupts, which, by definition, occur infrequently. This category can be further broken down into
     - Interrupts: peripheral devices and internal timers
     - System Service calls (**SYSCALL**)
     - TLB exceptions - exceptions related to the memory management unit (MMU)
     - Program Trap exceptions (e.g. Bus Error)

2. Using the data structures from Level 2 [Chapter 2], and the facility to handle both system service calls and device interrupts, timer interrupts in particular, provide a process scheduler – support multiprogramming.

Hence, the purpose of the Nucleus is to provide an environment in which asynchronous sequential processes (i.e. heavyweight threads) exist, each making forward progress as they take turns sharing the processor. Furthermore, the Nucleus provides these processes with exception handling routines, low-level synchronization primitives, and a facility for "passing up" the handling of Program Trap, TLB exceptions and certain **SYSCALL** requests to the Support Level. [Chapter 4]

Important Point: Since virtual memory is not supported until the Support Level, all addresses at this level are assumed to be physical addresses.

In summary, after some one-time Nucleus initialization code, the Nucleus will repeatedly dispatch a process, i.e. remove a *pcb* from the Ready Queue and perform a **LDST** on the processor state stored in the *pcb* (`p_s`). This *Current Process* will run until:

- It makes a system call (**SYSCALL**). The Nucleus will handle the system call or pass along the handling to the Support Level. Some system calls block the Current Process - the *pcb* is placed on the ASL and the Scheduler is called to dispatch the next job. If the system call is non-blocking, control is returned to the Current Process.

- It terminates; which is signaled via a system call. The Nucleus will call the Scheduler to dispatch the next process on the Ready Queue.

- The timer assigned to the Scheduler generates an interrupt; the Current Process's quantum/time slice has expired. Its *pcb* is enqueued back on the Ready Queue and the Scheduler is called to dispatch the next job.

- A device interrupt occurs (exclusive of the timer assigned to the Scheduler). The interrupt is acknowledged, and the device's status code is passed along to the *pcb* (i.e. process) that got unblocked as a result of the interrupt; the *pcb* that was waiting for the I/O to complete. The newly unblocked *pcb* is enqueued back on the Ready Queue and control is returned to the Current Process.

- If the Scheduler ever discovers that the Ready Queue is empty it will either **HALT** execution (no more processes to run), **WAIT** for an I/O to complete (which will unblock a *pcb* and populate the Ready Queue), or **PANIC** (halt execution in the presence of deadlock).

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

- Nucleus initialization. [Section 3.1]
- The Scheduler. [Section 3.2]
- **SYSCALL** processing. [Section 3.5]
- Device interrupt handler. [Section 3.6]
- The passing up of the handling of all other events. This includes TLB-Refill events [Section 3.3], **SYSCALL**s not handled at this level, page faults, Program Trap exceptions, etc. [Section 3.7]

3.1 Nucleus Initialization

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

1. Declare the Level 3 global variables. This should include:
   - ~~Process Count: integer indicating the number of started, but not yet terminated processes.~~
   - ~~Soft-block Count: A process can be either in the "ready," "running," or "blocked" (also known as "waiting") state. This integer is the number of started, but not terminated processes that in are the "blocked" state due to an I/O or timer request.~~
   - ~~Ready Queue: Tail pointer to a queue of *pcb*s that are in the "ready" state.~~
   - ~~Current Process: Pointer to the *pcb* that is in the "running" state, i.e. the current executing process.~~

- Device Semaphores: The Nucleus maintains one integer semaphore for each external (sub)device in μMPS3, plus one additional semaphore to support the Pseudo-clock. [Section 3.6.3]
  Since terminal devices are actually two independent sub-devices, the Nucleus maintains two semaphores for each terminal device. [Section **??**-*pops*]

2. Populate the Processor 0 Pass Up Vector. The Pass Up Vector is part of the BIOS Data Page, and for Processor 0, is located at 0x0FFF.F900. [Section **??**-*pops*]

   The Pass Up Vector is where the BIOS finds the address of the Nucleus functions to pass control to for both TLB-Refill events and all other exceptions. Specifically,

   - Set the Nucleus TLB-Refill event handler address to

     ```
     xxx->tlb_refll_handler =
                 (memaddr) uTLB_RefillHandler;
     ```

     where `memaddr`, in `types.h`, has been aliased to `unsigned int`.
     Since address translation is not implemented until the Support Level, `uTLB_RefillHandler` is a place holder function whose code is provided. [Section 3.3] This code will then be replaced when the Support Level is implemented.
   - Set the Stack Pointer for the Nucleus TLB-Refill event handler to the top of the Nucleus stack page: 0x2000.1000. Stacks in μMPS3 grow down.
   - Set the Nucleus exception handler address to the address of your Level 3 Nucleus function (e.g. `fooBar`) that is to be the entry point for exception (and interrupt) handling [Section 3.4].

     ```
     xxx->exception_handler = (memaddr) fooBar;
     ```

   - Set the Stack pointer for the Nucleus exception handler to the top of the Nucleus stack page: 0x2000.1000.

3. Initialize the Level 2 (phase 1 - see Chapter 2) data structures:

```
initPcbs()
initASH()       initNamespaces()
```

4. Initialize all Nucleus maintained variables: Process Count (0), Soft-block Count (0), Ready Queue (`mkEmptyProcQ()`), and Current Process (NULL). Since the device semaphores will be used for synchronization, as opposed to mutual exclusion, they should all be initialized to zero.

5. Load the system-wide Interval Timer with 100 milliseconds. [Section 3.6.3]

6. Instantiate a single process, place its *pcb* in the Ready Queue, and increment Process Count. A process is instantiated by allocating a *pcb* (i.e. `allocPcb()`), and initializing the processor state that is part of the *pcb*. In particular this process needs to have interrupts enabled, the processor Local Timer enabled, kernel-mode on, the **SP** set to RAMTOP (i.e. use the last RAM frame for its stack), and its **PC** set to the address of `test`. Furthermore, set the remaining *pcb* fields as follows:

   - Set all the Process Tree fields to NULL.
   - Set the accumulated time field (`p_time`) to zero.
   - Set the blocking semaphore address (`p_semAdd`) to NULL.
   - Set the Support Structure pointer (`p_supportStruct`) to NULL.

   Important Point: When setting up a new processor state one must set the *previous* bits (i.e. **IEp** & **KUp**) and not the *current* bits (i.e. **IEc** & **KUc**) in the **Status** register for the desired assignment to take effect after the initial **LDST** loads the processor state. [Section **??**-*pops*]

   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

```
yyy->p_s.s_pc = (memaddr) test;
```

Remember to declare *test* as "external" in your program by including the line:

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

For rather technical reasons, whenever one assigns a value to the **PC** one must also assign the same value to the general purpose register **t9**. (a.k.a. `s_t9` as defined in types.h.) [Section **??**-*pops*]

7. Call the Scheduler.

Once `main()` calls the Scheduler its task is complete since control should never return to `main()`. At this point the only mechanism for re-entering the Nucleus is through an exception; which includes device interrupts. As long as there are processes to run, the processor is executing instructions on their behalf and only temporarily enters the Nucleus long enough to handle a device interrupt or exception when they occur.

At boot/reset time the Nucleus is loaded into RAM beginning with the second frame of RAM: 0x2000.1000. The first frame of RAM is reserved for the Nucleus stack. Furthermore, Processor 0 will be in kernel-mode with all interrupts masked, and the processor Local Timer disabled. The **PC** is assigned 0x2000.1000 and the **SP**, which was initially set to 0x2000.1000 at boot-time, will now be some value less, due to the activation record for `main()` that now sits on the stack. [Section **??**-*pops*]

3.2 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.

Preemptive cpu scheduling requires the use of an interrupt generating system clock. μMPS3 offers two choices: the single system-wide Interval Timer or a processor's Local Timer (PLT). [Section **??**-*pops*]

One should use the PLT to support per processor scheduling since the Interval Timer is reserved for implementing Pseudo-clock ticks. [Section 3.6.3]

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

1. Remove the *pcb* from the head of the Ready Queue and store the pointer to the *pcb* in the Current Process field.

2. Load 5 milliseconds on the PLT. [Section **??**-*pops*]

3. Perform a Load Processor State (**LDST**) on the processor state stored in *pcb* of the Current Process (`p_s`).

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

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

1. If the Process Count is zero invoke the **HALT** BIOS service/instruction. [Section **??**-*pops*] Consider this a job well done!

2. If the Process Count > 0 and the Soft-block Count > 0 enter a *Wait State*. A Wait State is where the processor is not executing instructions, but "twiddling its thumbs" waiting for a device interrupt to occur. μMPS3 supports a **WAIT** instruction expressly for this purpose. [Section **??**-*pops*]

   Important Point: Before executing the **WAIT** instruction, the Scheduler must first set the **Status** register to enable interrupts **and** either disable the PLT (also through the **Status** register), or load it with a very large value. The first interrupt that occurs after entering a Wait State should not be for the PLT.

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

3.3 TLB-Refill events

As outlined above [Section 3.1], 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 (e.g. `uTLB_RefillHandler`)

The code for this function, for Level 3/Phase 2 testing purposes should be as follows:

```
void uTLB_RefillHandler () {
    setENTRYHI(0x80000000);
    setENTRYLO(0x00000000);
    TLBWR();
    LDST ((state_PTR) 0x0FFFF000);
}
```

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

3.4 Exception Handling

As described above [Section 3.1], at startup, the Nucleus will have populated the Processor 0 Pass Up Vector with the address of the Nucleus exception handler (`fooBar`) and the address of the Nucleus stack page (0x2000.1000). Therefore, if the Pass Up Vector was correctly initialized, `fooBar` 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). [Section **??**-*pops*]
The *cause* of this exception is encoded in the **.ExcCode** field of the **Cause** register (**Cause.ExcCode**) in the saved exception state. [Section **??**-*pops*]

- For exception code 0 (Interrupts), processing should be passed along to your Nucleus's device interrupt handler. [Section 3.6]
- For exception codes 1-3 (TLB exceptions), processing should be passed along to your Nucleus's TLB exception handler. [Section 3.7.3]
- For exception codes 4-7, 9-12 (Program Traps), processing should be passed along to your Nucleus's Program Trap exception handler. [Section 3.7.2]
- For exception code 8 (**SYSCALL**), processing should be passed along to your Nucleus's **SYSCALL** exception handler. [Section 3.5]

Hence, the entry point for the Nucleus's exception handling is in essence a case statement that performs a multi-way branch depending on the cause of the exception.

Important Point: To determine if the Current Process was executing in kernel-mode or user-mode, one examines the **Status** register in the saved exception state. In particular, examine the *previous* version of the **KU** bit (**KUp**) since the processor's exception handling circuitry will have performed a stack push on the **KU/IE** stacks in the **Status** register before the exception state was saved. [Section **??**-*pops*]

3.5 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**.

In particular, if the process making a **SYSCALL** request was in kernel-mode and **a0** contained a value in the range [1..10] then the Nucleus should perform one of the services described below.

3.5.1 Create_Process (SYS1)

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 SYS1 service continues to exist and to execute. If the new process cannot be created due to lack of resources (e.g. no more free *pcb*'s), an error code of -1 is placed/returned in the caller's **v0**, otherwise, return the value 0 in the caller's **v0**.

Good design calls for tight/strong cohesion and loose coupling between modules/classes/OS Levels, etc. Level 2 implements *pcb*s, and Level 3 utilizes queues of *pcb*s to create a basic multiprogramming environment. However, it is the Support Level that handles address translation as well as all exceptions beyond I/O interrupts and the first ten system calls (and then, only if in

kernel-mode). The design question then is how to provide Support Level access to *pcb* fields that will only be used in the Support Level.

The standard approach, at least in systems-level programming such as an OS, is to define a structure containing the additional Support Level fields (`support_t`) and then add a pointer (`support_t *`) to the *pcb*. The Support Level code needing access to these fields will execute a SYS8 [Section 3.5.8] which returns a pointer to the Current Process's `support_t` structure. This provides Support Level access to relevant *pcb* fields while hiding the Level 3 (and Level 2) *pcb* fields.

The SYS1 service is requested by the calling process by placing the value 1 in **a0**, a pointer to a processor state in **a1**, (optionally) a pointer to a Support Structure in **a2**, the pointer to the PID namespace of the new process in **a3**and then executing the **SYSCALL** instruction.

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

```
int pid = SYSCALL (CREATEPROCESS,
         state_t *statep, support_t * supportp, struct nsd_t *ns);
```

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 an unique identifier (PID) is returned to the Current Process. [Section 3.5.12]

This unique identifier can be the address of the PCB structure or a sequential value. The only constraints on this value are that this value must be not null (0) and there shall not be two processes with the same identifier.

In summary, for SYS1, one allocates a new *pcb* and initializes its fields:

- `p_s` from **a1**.
- `p_supportStruct` from **a2**. If no parameter is provided, this field is set to NULL.
- The *process queue* fields (e.g. `p_next`) by the call to `insertProcQ`
- The *process tree* fields (e.g. `p_child`) by the call to `insertChild`.
- `p_time` is set to zero; the new process has yet to accumulate any cpu time.

- `p_semAdd` is set to NULL; this *pcb*/process is in the "ready" state, not the "blocked" state.

- `namespace` is set to the namespace identified by the ns parameter, if ns is not NULL. It will inherit the namespace of the parent process otherwise.

3.5.2 Terminate_Process (SYS2)

This services causes the executing or a target process to cease to exist. [Section 3.9] In addition, recursively, all progeny of this process are terminated as well. Execution of this instruction does not complete until *all* progeny are terminated, after which the Scheduler should be called.

The SYS2 service is requested by the calling process by placing the value 2 in **a0** and then executing the **SYSCALL** instruction.

The following C code can be used to request a SYS2:

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

Where the mnemonic constant `TERMINATEPROCESS` has the value of 2.

If pid is 0, the calling process and its progeny is terminated, otherwise the process with the specified pid is terminated.

3.5.3 Passeren (P) (SYS3)

This service requests the Nucleus to perform a P operation on a semaphore.

The P or SYS3 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, or this process is blocked on the ASL (transitions from "running" to "blocked") and the Scheduler is called.

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

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

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

3.5.4 Verhogen (V) (SYS4)

This service requests the Nucleus to perform a V operation on a semaphore.

The V or SYS4 service is requested by the calling process by placing the value 4 in **a0**, the physical address of the semaphore to be V'ed in **a1**, and then executing the **SYSCALL** instruction.

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

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

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

3.5.5 Do_IO (SYS5)

Pandos 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 SYS5 for that device. Hence, a SYS5 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 values in **a1**and **a2**.

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 SYS5 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. [Section **??**-*pops*]

As discussed below [Section 3.6], the Nucleus will perform a V operation on the Nucleus maintained semaphore whenever that (sub)device generates an interrupt.

The SYS5 service is requested by the calling process by placing the value 5 in **a0**, the command address in **a1**, the command values pointer (that need to be written starting from the command address) in **a2**and then executing the **SYSCALL** instruction.

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

```
int ioStatus = SYSCALL (DOIO, int *cmdAddr,
         int *cmdValues, 0);
```

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

cmdValues is an array of 2 elements for the terminal devices, 4 elements for all the other devices.

At the completion of the I-O operation the device register values are copied back in the cmdValues array. The return value of this system call is 0 in case of success, -1 otherwise.

3.5.6 Get_CPU_Time (SYS6)

This service requests that the accumulated processor time (in microseconds) used by the requesting process be placed/returned in the caller's **v0**. Hence, the Nucleus records (in the *pcb*: `p_time`) the amount of processor time used by each process. [Section 3.8]

The SYS6 service is requested by the calling process by placing the value 6 in **a0** and then executing the **SYSCALL** instruction.

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

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

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

3.5.7 Wait_For_Clock (SYS7)

This service performs a P operation on the Nucleus maintained Pseudo-clock semaphore. This semaphore is V'ed every 100 milliseconds by the Nucleus. [Section 3.6.3]

Since the Pseudo-clock semaphore is a synchronization semaphore, this call should **always** block the Current Process on the ASL, after which the Scheduler is called. Hence, a SYS7 is used to transition the Current Process from the "running" state to a "blocked" state.

The SYS7 service is requested by the calling process by placing the value 7 in **a0** and then executing the **SYSCALL** instruction.

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

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

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

3.5.8 Get_SUPPORT_Data (SYS8)

This service requests a pointer to the Current Process's Support Structure. Hence, this service returns the value of `p_supportStruct` from the Current Process's *pcb*. If no value for `p_supportStruct` was provided for the Current Process when it was created, return NULL.

The SYS8 service is requested by the calling process by placing the value 8 in **a0** and then executing the **SYSCALL** instruction.

The following C code can be used to request a SYS8:

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

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

3.5.9 Get_Process_ID (SYS9)

This service requests the unique id of the process (PID). The PID of the parent need to take in consideration the PID namespace of a process, if a process is not in the same PID namespace of its parent, this systemcall, with the correct request of the parent PID, will return 0.

If the process is in the same PID namespace of its parent, this systemcall will return correctly the PID of the parent.

The SYS9 service is requested by the calling process by placing the value 9 in **a0** , TRUE in **a1** if we want the PID of the parent of the calling process, FALSE in **a1** if we want the PID of the calling process, and executing the **SYSCALL** instruction.

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

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

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

3.5.10 Get_children (SYS10)

This service returns the PIDs of the calling process' children belonging to the same namespace.

In fact, if a child process is not in the same PID namespace of the caller children, its PID is not added to the output array.

The SYS10 service is requested by the calling process by placing the value 10 in **a0** , an array of integers (which needs to be properly allocated) in **a1** , the array length in **a2** , and executing the **SYSCALL** instruction.

This service returns the total number of the process' children that has been or could have been added to the array. (if the return value is greater than the array length it means that some PIDs are missing).

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

```
int pid = SYSCALL (GETCHILDREN, int *children, int size, 0);
```

Where the mnemonic constant `GETCHILDREN` has the value of 10.

3.5.11 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 Program Trap exception response.

In particular the Nucleus should simulate a Program Trap exception when a privileged service is requested in user-mode. This is done by setting **Cause.ExcCode** in the stored exception state to *RI* (Reserved Instruction), and calling one's Program Trap exception handler.

Technical Point: As described above [Section 3.4], the saved exception state (for Processor 0) is stored at the start of the BIOS Data Page (0x0FFF.F000). [Section **??**-*pops*]

3.5.12 Returning from a SYSCALL Exception

For **SYSCALL**s calls that do not block or terminate, control is returned to the Current Process at the conclusion of the Nucleus's **SYSCALL** exception handler. Observe that the correct processor state to load (**LDST**) is the

saved exception state (located at the start of the BIOS Data Page [Section 3.4]) and not the obsolete processor state stored in the Current Process's *pcb*. The saved exception state was the state of the process at the time the **SYSCALL** was executed. The processor state in the Current Process's *pcb* was the state of the process at the start of it current time slice/quantum.

Hence, any return value described above (e.g. SYS6) needs to be put in the specified register in the stored exception state.

Furthermore, **SYSCALL**s that do not result in process termination (eventually) return control to the process's execution stream. This is done either immediately (e.g. SYS6) or after the process is blocked and eventually unblocked (e.g. SYS5). In any event the **PC** that was saved is, as it is for all exceptions, the address of the instruction that caused that exception – the address of the **SYSCALL** assembly instruction. Without intervention, returning control to the **SYSCALL** requesting process will result in an infinite loop of **SYSCALL**'s. To avoid this the **PC** must be incremented by 4 (i.e. the μMPS3 wordsize) prior to returning control to the interrupted execution stream. While the **PC** needs to be altered, there is no need, in this case, to make a parallel assignment to **t9**.

3.5.13 Blocking SYSCALLs

For **SYSCALL**s that block (SYS3, SYS5, and SYS7), a number of steps need to be performed:

- As described above [Section 3.5.12] the value of the **PC** must be incremented by 4 to avoid an infinite loop of **SYSCALL**s.
- The saved processor state (located at the start of the BIOS Data Page[Section 3.4]) must be copied into the Current Process's *pcb* (`p_s`).
- Update the accumulated CPU time for the Current Process. [Section 3.8]
- The Current Process is blocked on the ASL (`insertBlocked`), transitioning the process from the "running" state, to the "blocked" state.
- Call the Scheduler.

3.6 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 ⇒ 0xFFFF.FFFF transition.

Assuming that the (Processor 0) Pass Up Vector was properly initialized by the Nucleus as part of Nucleus initialization [Section 3.1], and that the Nucleus exception handler (`fooBar`) correctly decodes **Cause.ExcCode** [Section 3.4], control should be passed to one's Nucleus interrupt exception handler.

Which interrupt lines have pending interrupts is set in **Cause.IP**. [Section **??**-*pops*] 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. [Section **??**-*pops*]

Since Pandos is intended for uniprocessor environments only, interrupt line 0 may safely be ignored. [Chapter **??**-*pops*]

Note, many devices per interrupt line may have an interrupt request pending, and that many interrupt lines may simultaneously be on. Also, since each terminal device is two sub-devices, each terminal device may have two interrupts pending simultaneously as well. One should process only one interrupt at a time: the interrupt with the highest priority. The lower the interrupt line and device number, the higher the priority of the interrupt. When there are multiple interrupts pending, and the interrupt exception handler processes only the single highest priority pending interrupt, the interrupt exception handler will be immediately re-entered as soon as interrupts are unmasked again; effectively forming a loop until all the pending interrupts are processed.

Since terminal devices are actually two sub-devices, both sub-devices may have an interrupt pending simultaneously. For purposes of prioritizing pending interrupts, terminal transmission (i.e. writing to the terminal) is of higher priority than terminal receipt (i.e. reading from the terminal). Hence, the PLT (interrupt line 1) is the highest priority interrupt, while reading from terminal 7 (interrupt line 7, device 7; read) is the lowest priority interrupt.

The interrupt exception handler's first step is to determine which device or timer with an outstanding interrupt is the highest priority.

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

3.6.1 Non-Timer Interrupts

1. Calculate the address for this device's device register. [Section **??**-*pops*]
2. Save off the status code from the device's device register.
3. Acknowledge the outstanding interrupt. This is accomplished by writing the acknowledge command code in the interrupting device's device register. Alternatively, writing a new command in the interrupting device's device register will also acknowledge the interrupt.
4. Perform a V operation on the Nucleus maintained semaphore associated with this (sub)device. This operation should unblock the process (*pcb*) which initiated this I/O operation and then requested to wait for its completion via a SYS5 operation.
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: Perform a **LDST** on the saved exception state (located at the start of the BIOS Data Page [Section 3.4]).

Important Point: It is possible that the V operation (increment the indicated semaphore and unblock a *pcb*) returns NULL instead of a *pcb*. This can happen if while waiting for the initiated I/O operation to complete, an ancestor of this *pcb* was terminated. In this case, simply return control to the Current Process.

Important Point: It is also possible that there is no Current Process to return control to. This will be the case when the Scheduler executes the **WAIT** instruction instead of dispatching a process for execution. [Section 3.2]

Technical Point: In μMPS3 it is technically feasible for a process to initiate an I/O operation and for the interrupt associated with this operation to occur *before* it has an opportunity to execute its SYS5. However, the Pandos specification for the Support Level prevents this from happening.

3.6.2 Processor Local Timer (PLT) Interrupts

The PLT is used to support CPU scheduling. The Scheduler will load the PLT with the value of 5 milliseconds whenever it dispatches a process. [Section 3.2]

This "running" process will either:

- Terminate. Execute a SYS2 or cause an exception without having set a Support Structure address. [Section 3.7]
- Transition from the "running" state to the "blocked" state; execute a SYS3, SYS5, or SYS7.
- Be interrupted by a PLT interrupt.

The last option means that the Current Process has used up its time quantum/slice but has not completed its *CPU Burst*. Hence, it must be transitioned from the "running" state to the "ready" state.

The PLT portion of the interrupt exception handler should therefore:

- Acknowledge the PLT interrupt by loading the timer with a new value. [Section **??**-*pops*]
- Copy the processor state at the time of the exception (located at the start of the BIOS Data Page [Section **??**-*pops*]) 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.

3.6.3 The System-wide Interval Timer and the Pseudo-clock

The *Pseudo-clock* is a facility provided by the Nucleus for the Support Level. The Nucleus promises to perform a V operation, every 100 milliseconds, on a special Nucleus maintained semaphore; the Pseudo-clock semaphore. [Section 3.1]

This periodic V operation is called a *Pseudo-clock Tick*.

To perform a P operation on the Pseudo-clock semaphore (i.e. transition from the "running" state to the "blocked" state on this semaphore), the Current Process will perform a SYS7.

Since the Interval Timer is only used for this purpose, all line 2 interrupts indicate that it is time to P the Pseudo-clock semaphore; a Pseudo-clock tick.

The Interval Timer portion of the interrupt exception handler should therefore:

1. Acknowledge the interrupt by loading the Interval Timer with a new value: 100 milliseconds. [Section **??**-*pops*]

2. Unblock **ALL** *pcb*s blocked on the Pseudo-clock semaphore. Hence, the semantics of this semaphore are a bit different than traditional synchronization semaphores

3. Reset the Pseudo-clock semaphore to zero. This insures that all SYS7 calls block and that the Pseudo-clock semaphore does not grow positive.

4. Return control to the Current Process: Perform a **LDST** on the saved exception state (located at the start of the BIOS Data Page [Section 3.4]).

Important Point: It is also possible that there is no Current Process to return control to. This will be the case when the Scheduler executes the **WAIT** instruction instead of dispatching a process for execution. [Section 3.2]

3.7 Pass Up or Die

The Nucleus will directly handle all SYS1-SYS10 requests and device (internal timers and peripheral devices) interrupts. For all other exceptions (e.g. **SYSCALL** exceptions numbered 11 and above, Program Trap and TLB exceptions) the Nucleus will take one of two actions depending on whether the offending process (i.e. the Current Process) was provided a non-NULL value for its Support Structure pointer when it was created. [Section 3.5.1]

- If the Current Process's `p_supportStruct` is NULL, then the exception should be handled as a SYS2: 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."

When an exception occurs, the processor, in concert with the BIOS-Excpt handler, "passes up" the handling of the exception to the Nucleus: store the saved exception state at an accessible location known to the Nucleus, and pass control to a routine specified by the Nucleus, i.e. the Nucleus Exception handler (`fooBar`).

- The location, in this case, is fixed; a given location in the BIOS Data Page. (For Processor 0, this is 0x0FFF.F000) [Section **??**-*pops*]

- The address (and stack pointer) for the handler to pass control to was seeded by the Nucleus, during Nucleus initialization, in the appropriate location of the Pass Up Vector. [Section 3.1]

When the Nucleus "passes up" exception handling to the Support Level, it essentially performs the same two tasks: copy the saved exception state into a location accessible to the Support Level, and pass control to a routine specified by the Support Level.

There is only one location for the saved exception state and one Pass Up Vector for the Nucleus. This is because the Nucleus runs in single threaded mode with interrupts masked; hence with no concurrency. The Nucleus services run in a "one at a time" mode, and each invocation running to completion without interruption. Hence the reusability of the BIOS Data Page location for the saved exception state and Pass Up Vector. This is also why Nucleus services are so limited: do only what must be done in single threaded mode, and pass up the handling of all other service requests.

Since the Support Level runs in a fully concurrent mode (interrupts unmasked), each process needs its own location(s) for their saved exception states, and addresses to pass control to: The Support Structure.

Furthermore, the concurrency at the Support Level is not only interprocess, but intra-process as well. The Support Level, while handling a passed up **SYSCALL**, can trigger a page fault. For this reason, the Support Structure contains **two** locations for saved exception states, and two addresses for handlers. One `state_t`/PC address pair for:

- TLB exceptions (i.e. page faults): The Support Level TLB exception handler.

- All other exceptions: The Support Level general exception handler.

One last important detail. The Support Structure's version of a Pass Up Vector needs to contain three register values and not two. In addition to the **PC/SP**, one also needs a new value for the **Status** register.

A **PC/SP/Status** combination is also referred to as a *context*. Hence the Support Structure's version of a Pass Up Vector needs to store two processor context sets: one for non-TLB exceptions and one for TLB exceptions.

The following two structures are provided:

```
/* process context */
typedef struct context_t {
    /* process context fields */
    unsigned int  c_stackPtr,             /* stack pointer value */
                  c_status,               /* status reg value    */
                  c_pc;                   /* PC address          */
} context_t;

typedef struct support_t {
    int           sup_asid;               /* Process Id (asid)   */
    state_t       sup_exceptState[2];     /* stored excpt states */
    context_t     sup_exceptContext[2];   /* pass up contexts    */
    ... other fields to be added later
} support_t;

 /* Exceptions related constants */
#define PGFAULTEXCEPT 0
#define GENERALEXCEPT 1
```

To pass up the handling of an exception:

- Copy the saved exception state from the BIOS Data Page to the correct `sup_exceptState` field of the Current Process. The Current Process's *pcb* should point to a non-null `support_t`.
- Perform a **LDCXT** using the fields from the correct `sup_exceptContext` field of the Current Process. [Section **??**-*pops*]

3.7.1 SYSCALL Exceptions Numbered 11 and Above

A **SYSCALL** exception numbered 11 and above occurs when the Current Process executes the **SYSCALL** instruction (**Cause.ExcCode** is set to 8

[Section 3.4]) and the contents of **a0** is greater than or equal to 11.

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

3.7.2 Program Trap Exception Handling

A Program Trap exception occurs when the Current Process attempts to perform some illegal or undefined action. A Program Trap exception is defined as an exception with **Cause.ExcCode**s of 4-7, 9-12. [Section 3.4]

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

3.7.3 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.ExcCode**s of 1-3. [Section 3.4]

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

3.8 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 V the Pseudo-clock semaphore. The Interval Timer is reserved for this purpose.

The third timing need is that the Nucleus is tasked with keeping track of the accumulated CPU time used by each process. [Section 3.5.6]

A field has been defined in the *pcb* for this purpose (`p_time`). Hence SYS6 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. [Section **??**-*pops*]

The three timer devices are mechanisms for implementing Pandos'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? Note: it is possible for an I/O or Interval Timer interrupt to occur even when there is no Current Process.

- While the time spent by the Nucleus handling a **SYSCALL** request needs to be measured for Pseudo-clock tick and quantum/time slice purposes, which process, if any, should be "charged" with this time?

It is important to understand the functional differences between the three μMPS3 timer devices. This includes, but is not limited to understanding that the TOD clock counts up while the other two timers count down, and that the behavior of the PLT differs from that of the Interval Timer. The PLT can be enabled/disabled via the processor Local Timer enable bit (**Status.TE**). [Section **??**-*pops*]

3.9 Process Termination

When a process is terminated (SYS2 or the "Die" portion of Pass Up or Die) there is actually a whole (sub)tree of processes that get terminated. There are a number of tasks that must be accomplished:

- The root of the sub-tree of terminated processes must be "orphaned" from its parents; its parent can no longer have this *pcb* as one of its progeny (`outChild`).

- If the value of a semaphore is negative, it is an invariant that the absolute value of the semaphore equal the number of *pcb*'s blocked on that semaphore. Hence if a terminated process is blocked on a semaphore, the value of the semaphore must be adjusted; i.e. incremented.

- 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 (and hence incremented) by the interrupt handler.
- The process count and soft-blocked variables need to be adjusted accordingly.
- Processes (i.e. *pcb*'s) can't hide. A *pcb* is either the Current Process ("running"), sitting on the Ready Queue ("ready"), blocked on a device semaphore ("blocked"), or blocked on a non-device semaphore ("blocked").

3.10 Nuts and Bolts

3.10.1 Module Decomposition

One possible module decomposition is as follows:

1. initial.c This module implements `main()` and exports the Nucleus's global variables. (e.g. process count, device semaphores, etc.)
2. interrupts.c This module implements the device/timer interrupt exception handler. This module will process all the device/timer interrupts, converting device/timer interrupts into V operations on the appropriate semaphores.
3. exceptions.c This module implements the TLB, Program Trap, and **SYSCALL** exception handlers. Furthermore, this module will contain the provided skeleton TLB-Refill event handler (e.g. `uTLB_RefillHandler`).
4. scheduler.c This module implements the Scheduler and the deadlock detector.

3.10.2 Accessing the `libumps` Library

Accessing the **CP0** registers and the BIOS-implemented services/instructions in C (e.g. **WAIT**, **LDST**) is via the `libumps` library. [Chapter **??**-*pops*] Simply include the line

```
#include ``/usr/include/umps3/umps/libumps.h''
```

in one's source files.[1]

3.11 Testing

There is a provided test file, p2test.c that will "exercise" your code. [Appendix A]

As with any non-trivial system, you are strongly encouraged to use the *make* program to maintain your code. A sample *Makefile* has been supplied. See Chapter **??** in the POPS reference for more compilation details.

Once your (seven?) source files (two from Phase 1 and four from Phase 2) have been correctly compiled, linked together (with appropriate linker script, `crtso.o`, and `libumps.o`), and post-processed with umps3-elf2umps (all performed by the sample *Makefile*), your code can be tested by launching the μMPS3 emulator. At a terminal prompt, enter:

```
umps3
```

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. [Chapter **??**]

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 either be System Halted for successful termination, or Kernel Panic for unsuccessful termination.

[1] The file libumps.h is part of the μMPS3 distribution.
/usr/include/umps3/umps/ is the recommended installation location for this file.