Chapter 3 - Phase 2

Source by Michael Goldweber, updated by Peter Rozzi

Phase 2 builds on many tools that have been defined in ROM. You will create the exception handlers which the ROM will redirect to. We have four types of exceptions: Program Traps (prgrmTrap), System Calls (syscalls), TLB Management, and Interrupts (interrupts have a unique handler because we need to deal with devices). You will also create a scheduler to facilitate the running of processes with the structures defined in phase 1.

Ultimately, phase 2's purpose is to organize asynchronous sequential processes (represented as PCBs), and handle exceptions. We will manipulate the addresses that enable/disable virtual memory, but JaeOS will not support virtual memory until phase 3 - so all addresses in phase 2 are assumed to be physical addresses.

3.1 NUCLEUS INITIALIZATION

The initialization function serves as the main() function of JaeOS which populates all fields and allocates a starting process.

Before entering the initialization function, you should define the following
readyQueue - a tail pointer to a queue of PCBs, which represents processes that are ready to be executed
currentProcess - a pointer to a PCB that represents the currently executing process
processCount - the number of total active processes

softBlockedCount - the number of processes currently waiting on a device (either I/O or the clock)

1. Initialize all global global phase 2 variables (e.g. processCount, softBlockedCount, readyQueue, currentProcess, and any you might add later)
2. Initalize the phsae 1 data structures with initPCBs() and initASL()
3. Initialize a number of device semaphores to zero. There are 8 devices per line(device type) except for line 7(Terminals) for which there are 16 (8 Read, 8 Write). Finally, the pseudo-clock also has a single semaphore. Lines 0 and 1 are not currently used by JaeOS, and there is only one device on line 2(the clock) which does not require a semaphore. So, we have (8 * 5) + 16 + 1 = 49 semaphores
4. Populate the new areas for exception handling in the kernel reserved frame
    a. Set the PC of each area to its respective handler
    b. Set the SP to RAMPTOP - the exception handlers will each use the last frame of RAM as their stack
    c. Set the CPSR to disable interrupts and be in kernel mode
    d. Set the CP15 register to turn virtual memory off
5. Allocate a process and place it on the readyQueue. Similar to the new area states, the registers in the starting process' state need to be populated
    a. Set the PC to the address of a starting function
       i.e. PC = (unsigned int)test
    b. Set the SP to RAMTOP - FRAMESIZE (the second to last RAM frame)

c. Set the CPSR register to enable interrupts and be in kernel mode
d. Set the CP15 register to turn virtual memory off

6. Finally, call the scheduler

Once the scheduler has been called, control will never return to the initialization function.

3.2 THE SCHEDULER

The scheduler facilitates the running of all processes on the ready queue. This section explains the implementation of a round-robin scheduler with a timer of 5 milliseconds (5000 microseconds).
The scheduler also performs deadlock detection and invokes a kernel PANIC if such a situation occurs.

The scheduler begins by checking the state of the ready queue
We execute the following steps if the readyQueue is empty:

1. If the processCount = 0, invoke the HALT instruction.
2. If the processCount > 0 and softBlockedCount = 0 this indicates a state of deadlock, so we invoke the PANIC instruction.
3. If the processCount > 0 and softBlockedCount > 0 this indicates that there are processes currently waiting on devices, so we enable interrupts and invoke the WAIT instruction.

If the readyQueue was not empty, then we retrieve the next process, set it to the currentProcess, set the timer for 5000 microseconds, and load the state of the currentProcess.

3.3 SYSTEM CALL EXCEPTION HANDLING

A system call occurs when an SVC or BKPT assembler instruction is executed. If the new areas in the kernel reserved frame were properly initialized, then these exceptions will be redirected to your syscall handler. The executing process will place relevant variables in a1-a4, and then execute the requested syscall.

The requested syscall can be found in a1

If the executing process was in kernel mode, and the value in a1 is within the legal range (1-8), then we perform one of the syscalls detailed below.

3.3.1 - Create Process (SYS1)

The ‘create process’ system call is executed when the requested syscall = 1

Registers used:

A1 - to return the outcome of the syscall (FAILURE (-1) or SUCCESS (0))

A2 - should contain the physical address of a processor state. This state will be used as the initial state for the new process

When executed, this syscall creates a new process, **makes it a child of the current process,** and then places the state held in a2 into the new process. If the process could not be created for any reason (e.g. if there are

no free PCBs), we return a FAILURE code (-1) in a1, otherwise return a SUCCESS code (0). Finally, we load back into the current process

3.3.2 - Terminate Process (SYS2)

The terminate process syscall is executed when the requested syscall = 2
Registers used: None

When executed, this syscall begins to recursively delete the current process and all of its children. This syscall finishes only after the current process and every member of its child tree have been terminated.

The recursive step of SYS2 must remove any references to each PCB, and maintain the processCount, softBlockedCount, and semaphore variables. The recursive call should execute as follows:

1. If the given process has a child, execute this function on that child (i.e. the recursive step).
2. If the given process is the current process, orphan it. (i.e. remove it from any parent it might have).
3. If the given process is on the readyQueue, remove it from the readyQueue.
4. If the given process is blocked on a semaphore, we have 2 cases:
    a. If it is blocked on a device semaphore, decrement softBlockedCount.
    b. If it is blocked on a non-device, increment that semaphore.
5. Free the given process' PCB, and decrement processCount.

Outside of the recursive call, the current process must be set to NULL. Finally, we give control to the scheduler.

In step 4a, we do not increment the process' semaphore because an interrupt will inevitably occur and the semaphore will be incremented then.

3.3.3 - Verhogen (V/Signal) (SYS3)

The verhogen syscall is executed when the requested syscall = 3
Registers used: A2 - physical address of the semaphore to be V'd

Retrieve the physical address of the semaphore in a2 and then increment its value.
If the semaphore is now <= 0, unblock a process from it and place that process on the readyQueue.

Finally, whether this was an unblocking signal or not, load back to the currentProcess.

3.3.4 - Passeren (P/Wait) (SYS4)

The passeren syscall is executed when the requested syscall = 4
Registers used: A2 - contains the physical address of the semaphore to be P'd
Retrieve the physical address of the semaphore, and then **decrement** its value.

If the semaphore value is now < 0, block the currentProcess on it and then call the scheduler, otherwise, load back to the currentProcess.

3.3.5 - Specify Exception State Vector (SYS5)

The ‘specify exception state vector’ syscall is executed when the requested syscall = 5
Registers used:
A2 - the type of exception for which
A3 - the address into which the old processor state is to be stored (i.e. prgrmTrap/Sys/TLB old area)
A4 - the address that is to be taken as the new processor state if an exception occurs (i.e. the prgrmTrap/Sys/TLB new area)

The exception types are as follows:

- 0 is a TLB exception
- 1 is prgrmTrap exceptions
- 2 is SysCall exceptions

Save the contents of A3 and A4 in the invoking process’ PCB so that the respective exception may be “passed up” and handled by the passUpOrDie function (explained in section 3.6).

Each process may request a SYS5 only one time for each exception type. If a process attempts to call SYS5 on the same exception type more than once, we execute SYS2 to kill the process and its children.

3.3.6 - Get CPU Time (SYS6)

The ‘get CPU time’ syscall is executed when the requested syscall = 6
Registers used: A1 - to return the processor time (in microseconds)

This syscall simply places the amount of time (in microseconds) used by the process into A1. This means that we must also record the time used by the current process in its PCB.

3.3.7 - Wait For Clock (SYS7)

The ‘wait for clock’ syscall is executed when the requested syscall = 7
Registers used: None

This instruction simulates a P (wait) operation on the pseudo-clock timer semaphore. This semaphore must also be signalled every 100 milliseconds by the kernel by the interrupt handler.

3.3.8 - Wait for I/O Device (SYS8)

The ‘wait for I/O device’ syscall is executed when the requested syscall = 8
Registers used:
A1 - device status to be returned
A2 - interrupt line number
A3 - device number
A4 - read or write operation for terminals (1 for read, 0 for write)

SYS8 performs a P (wait) operation on a device semaphore, determined by the values placed in A3, A3, and A4. The device semaphore can be determined by the formula:

*8 * (lineNumber - 3) + deviceNumber*

We subtract 3 because there are no devices on the first 3 lines.
Terminals are two independent sub-devices, and are handled by the interrupt handler as two independent devices (hence each terminal has two semaphores, one for receiving and one for transmitting), but are all on the same line. So, if the device is a **transmitting terminal**, modify the formula as follow:

*8 * (lineNumber - 3) + deviceNumber* ***+ 8***

A signal operation will be performed on a device semaphore whenever that device triggers an interrupt, once the interrupt has been handled and the process resumes execution, the device's status can be found in A1.
It is possible that an interrupt could occur prior to the SYS8 call. In this case, the calling process will not block as a result of the P operation in SYS8, and the interrupting device's status, which was stored, can now be placed in A1 prior to resuming execution.

3.3.9 - SYS1 - SYS8 in User Mode

The eight syscalls defined above are considered privileged instructions, and their execution should only be allowed for processes running in kernel mode.
Specifically, if any of these syscalls are requesting by a process running in user mode, we should simulate a prgrmTrap exception. This is done by executing the following steps:

1. Copy the processor state from the **sysCall old area** into the **prgrmTrap old area**
2. Set the **CP15.Cause** in the **prgrmTrap old area** to *Reserved Instruction (*20)
3. Call the prgrmTrap exception handler

3.3.10 - SYS9 and above

If the requested syscall >= 9
The SysCall handler will directly handle all syscalls 1-8. Syscalls above 8 are considered exceptions.
We handle them by calling the passUpOrDie function with an error code of 2 (explained in section 3.6)

3.4 prgrmTrap Exception Handling

A prgrmTrap (program trap) exception occurs when the executing process attempts to perform some illegal or undefined action.
We handle them by calling the passUpOrDie function with an error code of 1 (explained in section 3.6)

3.5 TLB Exception Handling

A TLB exception occurs when μARM fails an attempt to translate a virtual address into its physical address.

We handle TLB exceptions by calling the passUpOrDie function with an error code of 0 (explained in section 3.6)

3.6 passUpOrDie

When the operating system encounters a prgrmTrap exception, TLB exception, or SYS exception, it should execute the passUpOrDie function with the proper error code as a parameter

- 0 for a TLB exception
- 1 for a prgrmTrap exception
- 2 for a SysCall exception

Based on the error code, this function will either “pass up” to a secondary exception handler, or kill the offending process.
If a SYS5 was called for the respective exception, then we “pass up” to the new handler by loading the state defined by SYS5.
If a SYS5 was *not* called for the respective exception, then we perform a SYS2 to kill the offending process and all of its children.

3.7 Interrupt Handling

A device intterupt occurs when an I/O request completes, or when the interval timer transitions from 0x00000000 to 0xFFFFFFFF.
The pending interrupt lines are set in CP15.Cause, and for line 3-7, the interrupting devices bit map will indicate which devices on these interrupt

lines have a pending interrupt. Since JaeOS is intending only for uniprocessor environment, lines 0 and 1 may be safely ignored, and line 2 only has a single device (the interval timer).
It is possible that multiple devices from multiple lines could have pending interrupts, so the handler will only handle interrupts with the highest priority at the time of execution. The lower the line and device numbers, the higher the priority (e.g. Line 2 is the highest priority device, follwed by line 3, device 0).
For the purposes of interrupt handling, **terminal transmission is considered higher priority than terminal receipt**.

The interrupt handler operate as follows:

1. Determine the line number of the pending interrupt.
    a. If the line number is 2, simulate a V operation on all processes waiting on the interval timer's semaphore. If we have a current process, load it, otherwise call the scheduler.
2. If the line number was not 2, use the pending interrupts bitmap to find the interrupting device number.
3. Determine the location of the device's register using the formula:
    0x00000040 + ((lineNum - 3) * 16)
4. Copy the status field from the device register, and set the command field to ACK (1).
5. Signal the device's semaphore and decrement softBlockedCount if a process was unblocked.
6. Finally, reload the current process. If there is no current process, call the scheduler.

3.8 Nuts and Bolts

3.8.1 Timing Issues

uarm only has one clock, but it must accomplish two distinct timing needs:

1. Generate an interrupt to signal the end of processes' time slices.
2. Generate an interrupt at 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) before the end of their current time slice. A more careful accounting method is called for; one where some (most) of the Processor Timer's interrupts represent the conclusion of a time slice while others represent the conclusion of a pseudo-clock tick.

When no process requests a SYS7, if left unadjusted, the pseudo-clock timer semaphore 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 SVC 7 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 processor time used by each process must also be kept track of (i.e. SYS6). This implies an additional field to be added to the PCB 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 timer device is a mechanism for implementing JaeOS's scheduling policies.
The time spent by handling interrupts and syscalls needs to be measured for the pseudo-clock tick, but which process if any should be "charged" for this time?

3.8.2 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. scheduler.c This module implements JaeOS' process scheduler and deadlock detector.
3. exceptions.c This module implements the TLB prgrmTrap and SysCall exception handlers.
4. 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.

3.8.3 Accessing the libuarm Library

Accessing the CP15 registers and the ROM-implemented services/instructions in C is via the libuarm library.

Simply include the line:

#include "/usr/include/uarm/libuarm.h"

The file *libuarm.h* is part of the µARM distribution.

Make sure you know where it is installed in your local environment and alter this compiler directive appropriately.

3.8.4 Testing

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

You should individually compile all the source files from both phase1 and phase2 in addition to the phase2 test file using the command:

arm-none-eabi-gcc -mcpu=arm7tdmi -c FILENAME.c

The object files from Phases 1 & 2 should then be linked together using the command:

arm-none-eabi-ld
-T /usr/include/uarm/ldscripts/elf32ltsarm.h.uarmcore.x
-o kernel.core.uarm
p2test.o interrupts.o exceptions.o ....o*
/usr/include/uarm/crtso.o
/usr/include/uarm/libuarm.o30

which produces the file: *kernel.core.uarm*

The files *elf32ltsarm.h.uarmcore.x, crtso.o*, and *libuarm.o* are part of the μARM distribution. */usr/include/uarm/* are the recommended installation locations for these files.

Make sure you know where they are installed in your local environment and alter this command appropriately. The order of the object files in this command is important: specifically, the first two support files must be in their respective positions.

Finally, your code can be tested by launching uarm. Entering:

uarm

without any parameters loads the file *kernel.core.uarm* by default.

Hopefully the above will illustrate the benefits for using a Makefile to automate the compiling, linking, and converting of a collection of source files into a μARM executable file.

The test program reports on its progress by writing messages to TERMINAL0.

At the conclusion of the test program, either successful or unsuccessful, μARM 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.