Student Guide to the JaeOS Operating System Project

The Virtual Square Lab

Michael Goldweber
Xavier University

Renzo Davoli
University of Bologna

JaeOS, μARM are products of the Virtual Square Lab.
See `virtualsquare.org/` and `wiki.virtualsquare.org/`.

JaeOSv1.0

Contents

List of Figures

Preface

In my junior year as an undergraduate I took a course titled, "Systems Programming." The goal of this course was for each student to write a small, simple multitasking operating system, in S/360 assembler, for an IBM S/360. The students were given use of a machine emulator, Assist-V, for the development process. Assist, was a S/360 assembler programming environment. (Think SPIM for the 70's.) Assist-V was an extension of Assist that supported privileged instructions in addition to various emulated "attached" devices. The highlight of the course was if your operating system ran correctly (or at least without discernible errors), you would be granted the opportunity, in the dead of night, to boot the University's mainframe, an IBM S/370, with your operating system. (Caveat: The University used VM, IBM's virtual machine technology. Hence students didn't actually boot the whole machine with their OS's, just one VM partition. Nevertheless, booting/running a VM partition and booting/running the whole machine are isomorphic tasks.) No question, booting and running a handful of tasks concurrently on the University's mainframe with my own OS was one of highlights of my undergraduate education!

For my senior project I undertook to update Assist-V to the S/370 ISA. Since neither Assist nor Assist-V supported floating point instructions, this basically meant adding virtual memory support to Assist-V. I recall my surprise in the mid-1980's receiving an email from some institution that was still using Assist-V/370 to support their operating systems course.

My experience of writing a complete operating system repeated itself in graduate school. In this case the machine emulator was the Cornell Hypothetical Instruction Processor (CHIP); a made up architecture that was a cross between a PDP-11 and an IBM S/370. The operating system design was a three phase/layer affair called HOCA by its creator. While there was no real machine to test with, the thrill and sense of accomplishment of successfully completing the task, to say nothing of the many lessons learned throughout the experience were no less than the earlier experience.

In the late 1990's Professor Renzo Davoli and one of his graduate students Mauro Morsiani, in the spirit of both Assist-V/370 and CHIP, created MPS, a MIPS 3000 machine emulator that not only authentically emulated the processor (still no floating point), but also faithfully emulated five different device categories. Furthermore, they updated the HOCA project, which they called TINA, for this new architecture. Once again, students could take their operating system, developed and debugged on MPS (which also contained an excellent debugging facility) and run it unchanged on a real machine.

MPS, with its faithful emulation of the MIPS 3000, though, proved to be too complex for a one semester undergraduate project. Hence Renzo and myself set out to create μMPS – a pedagogically appropriate machine emulator appropriate for use by undergraduates. In addition, we updated TINA for this new architecture. That project is called Kaya.

μMPS and Kaya were originally released in 2004. Kaya was later updated and first published in its current form in 2009. In 2011 μMPS was updated by Tomislav Jonjic to μMPS2 with a new GUI and multiprocessor support.

As time passed, and as the ARM architecture grew in relevance, the MIPS architecture lost favor in the technical community. While μMPS2 and Kaya remain great projects, the demand for an OS project based on a current or industry relevant architecture grew. In 2015 Professor Davoli and a different graduate student, Marco Melletti created μARM – the ARM7tdmi equivalent to μMPS. Kaya was similarly updated and is now called JaeOS.

A raw machine emulator, such as μARM, which is fully described in the "μARM Principles of Operation," can support a wide variety of undergraduate, and graduate-level projects. The JaeOS project is just one such project. The Virtual Square Lab, which produced both JaeOS and μARM is also currently producing additional projects for μARM as well as for VDE, the Virtual Distributed Ethernet tool also produced by the Virtual Square Lab.

Finally Renzo and I wish to offer our heartfelt thanks all our students who have been our guinea pigs as we work through the bugs in in these systems. In particular we wish to thank:

- Mauro Morsiani who generously donated his time to modify MPS into μMPS.
- Tomislav Jonjic who extended μMPS to create the backward compatible μMPS2.
- Marco Melletti who created μARM.

- Jacob Wagner and Alex Fuerst who assisted greatly in the drafting of this document.

We wish to also thank our wives, Alessandra and Mindy and our children, without whose inexhaustible patience projects such as this would never see the light of day.

Michael Goldweber
August, 2017

1 Introduction

The JaeOS described below is similar to the T.H.E. system outlined by Dijkstra back in 1968[4]. Dijkstra's paper described an OS divided into six layers. Each layer i was an ADT or abstract machine to layer $i + 1$; successively building up the capabilities of the system for each new layer to build upon. The OS described here also contains six layers, though the final OS is not as complete as Dijkstra's.

JaeOS is actually the latest instantiation of an older "learning" operating system design. Ozalp Babaoglu and Fred Schneider originally described this operating system, calling it the HOCA OS[3], for implemention on the Cornell Hypothetical Instruction Processor (CHIP)[2, 1]. Later, Renzo Davoli and Mauro Morsiani reworked HOCA, calling it TINA[8] and ICAROS[7], for implementation on the Microprocessor (without) Pipeline Stages (MPS)[9, 8]. In 2004 TINA evolved into Kaya for the μMPS and μMPS2 emulators[6, 5]. JaeOS is a port of Kaya for μARM.

Level 0: The base hardware of μARM.

Level 1: The additional services provided in ROM. This includes the services provided by the ROM-Excpt handler (i.e. processor state save and load), the ROM-TLB-Refill handler (i.e. searching PTE's for matching entries and loading them into the TLB), and the additional ROM services/instructions.

The μARM Principles of Operation contains a complete description of both Level 0 and 1.

Level 2: The Queues Manager (Phase 1 – described in Chapter 2). Based on the key operating systems concept that active entities at one layer are just data structures at lower layers, this layer supports the management of queues of structures; ProcBlk's.

Level 3: The Kernel (Phase 2 – described in Chapter 3). This level implements eight new kernel-mode process management and synchronization primitives in addition to multiprogramming, a process scheduler, device interrupt handlers, and deadlock detection.

Level 4: The Support Level (Phase 3 – described in Chapter **??**). Level 3 is extended to support a system that can support multiple user-level cooperating processes that can request I/O and which run in their own virtual address space. Furthermore, this level adds user-level synchronization, and a process sleep/delay facility.

Level 5: The File System (Phase 4) This level implements the abstraction of a flat file system by implementing primitives necessary to create, rename, delete, open, close, and modify files.

Level 6: The Interactive Shell – why not?

1.1 Notational conventions

- Words being defined are *italicized*.
- Register, fields and instructions are **bold**-marked.
- Field **F** of register **R** is denoted **R**. **F**
- Bits of storage are numbered right-to-left, starting with 0.
- The i-th bit of a storage unit named **N** is denoted **N**[i].
- Memory addresses and operation codes are given in hexadecimal and displayed in big-endian format.

- All diagrams illustrate memory and going from low addresses to high addresses using a left to right, bottom to top orientation.
- References to the μARM Principles of Operation will have a pops suffix. e.g. A reference to the chapter on Exception Handling will be denoted: Section 2.4 - pops.

UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity.

Dennis Ritchie

2

Phase 1 - Level 2: The Queues Manager

Level 2 of JaeOS instantiates the key operating system concept that active entities at one layer are just data structures at lower layers. In this case, the active entities at a higher level are processes (i.e. programs in execution) and the data structure(s) that represent them at this level are *process control blocks* (ProcBlk's).

```
/* process control block type */
typedef struct pcb_t {
    /* process queue fields */
    struct pcb_t        *p_next,     /* pointer to next entry */

    /* process tree fields */
                        *p_prnt,     /* pointer to parent     */
                        *p_child,    /* pointer to 1st child  */
                        *p_sib;      /* pointer to sibling    */

    state_t             p_s;         /* processor state */
    int                 *p_semAdd;   /* pointer to sema4 on   */
                                     /* which process blocked */

    /* plus other entries to be added later */
} pcb_t;
```

The queue manager will implement four ProcBlk related functions:

- The allocation and deallocation of ProcBlk's.
- The maintenance of queues of ProcBlk's.
- The maintenance of trees of ProcBlk's.
- The maintenance of a single sorted list of *active semaphore descriptors*, each of which supports a queue of ProcBlk's.

2.1 The Allocation and Deallocation of ProcBlk's

One may assume that JaeOS supports no more that *MAXPROC* concurrent processes; where MAXPROC should be set to 20 (in the file `const.h`). Thus this level needs a "pool" of MAXPROC ProcBlk's to allocate from and deallocate to. Assuming that there is a set of MAXPROC ProcBlk's, the free or unused ones can be kept on a NULL-terminated single linearly linked list (using the `p_next` field), called the *pcbFree* List, whose head is pointed to by the variable `pcbFree_h`.

To support the allocation and deallocation of ProcBlk's there should be the following three externally visible functions:

- ProcBlk's which are no longer in use can be returned to the pcbFree list by using the method:
  `void freePcb(pcb_t *p)`

  /* Insert the element pointed to by `p` onto the pcbFree list. */

- ProcBlk's should be allocated by using:
  `pcb_t *allocPcb()`

  /* Return NULL if the pcbFree list is empty. Otherwise, remove an element from the pcbFree list, provide initial values for ALL of the ProcBlk's fields (i.e. NULL and/or 0) and then return a pointer to the removed element. ProcBlk's get reused, so it is important that no previous values persist in a ProcBlk when it gets reallocated. */

There is still the question of how one acquires storage for MAXPROC ProcBlk's and gets these MAXPROC ProcBlk's initially onto the pcbFree list. Unfortunately, there is no `malloc()` feature to acquire dynamic (i.e. non-automatic) storage that will persist for the lifetime of the OS and not just the lifetime of the function they are declared in. Instead, the storage for the MAXPROC ProcBlk's will be allocated as *static* storage. A static array of MAXPROC ProcBlk's will be declared in `initPcbs()`. Furthermore, this method will insert each of the MAXPROC ProcBlk's onto the pcbFree list.

- To initialize the pcbFree List:
  `void initPcbs()`

  /* Initialize the pcbFree list to contain all the elements of the static array of MAXPROC ProcBlk's. This method will be called only once during data structure initialization. */

2.2 Process Queue Maintenance

The methods below do not manipulate a particular queue or set of queues. Instead they are generic queue manipulation methods; one of the parameters is a pointer to the queue upon which the indicated operation is to be performed.

The queues of ProcBlk's to be manipulated, which are called *process queues*, are all single circularly linked, via the `p_next` pointer field, lists. Instead of a head pointer, each queue will be pointed at by a tail pointer. One may optionally wish to make all these queues double circularly linked lists for greater efficiency. (This requires adding a `p_prev` pointer to the `pcb_t` structure definition.)

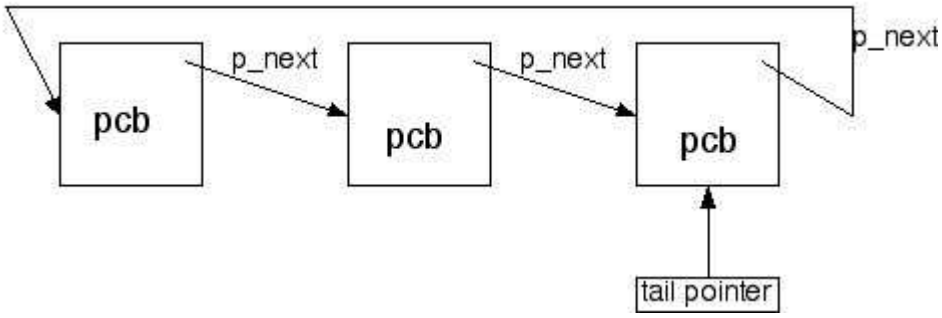

Figure 2.1: Process Queue

To support process queues there should be the following externally visible functions:

```
pcb_t *mkEmptyProcQ()
```

/* This method is used to initialize a variable to be tail pointer to a process queue.
Return a pointer to the tail of an empty process queue; i.e. NULL. */

```
int emptyProcQ(pcb_t *tp)
```

/* Return TRUE if the queue whose tail is pointed to by `tp` is empty. Return FALSE otherwise. */

```
void insertProcQ(pcb_t **tp, pcb_t *p)
```

/* Insert the ProcBlk pointed to by `p` into the process queue whose tail-pointer is pointed to by `tp`. Note the double indirection through `tp` to allow for the possible updating of the tail pointer as well. */

```
pcb_t *removeProcQ(pcb_t **tp)
```

/* Remove the first (i.e. head) element from the process queue whose tail-pointer is pointed to by `tp`. Return NULL if the process queue was initially empty; otherwise return the pointer to the removed element. Update the process queue's tail pointer if necessary. */

```
pcb_t *outProcQ(pcb_t **tp, pcb_t *p)
```

> /* Remove the ProcBlk pointed to by `p` from the process queue whose tail-pointer is pointed to by `tp`. Update the process queue's tail pointer if necessary. If the desired entry is not in the indicated queue (an error condition), return NULL; otherwise, return `p`. Note that `p` can point to any element of the process queue. */

```
pcb_t *headProcQ(pcb_t *tp)
```

> /* Return a pointer to the first ProcBlk from the process queue whose tail is pointed to by `tp`. Do not remove this ProcBlkfrom the process queue. Return NULL if the process queue is empty. */

2.3 Process Tree Maintenance

In addition to possibly participating in a process queue, ProcBlk's are also organized into trees of ProcBlk's, called *process trees*. The `p_prnt`, `p_child`, and `p_sib` pointers are used for this purpose.

The process trees should be implemented as follows. A parent ProcBlk contains a pointer (`p_child`) to a NULL-terminated single linearly linked list of its child ProcBlk's. Each child process has a pointer to its parent ProcBlk (`p_prnt`) and possibly the next child ProcBlk of its parent (`p_sib`). For greater efficiency you may want to make the linked list of child ProcBlk's a NULL-terminated double linearly linked list.

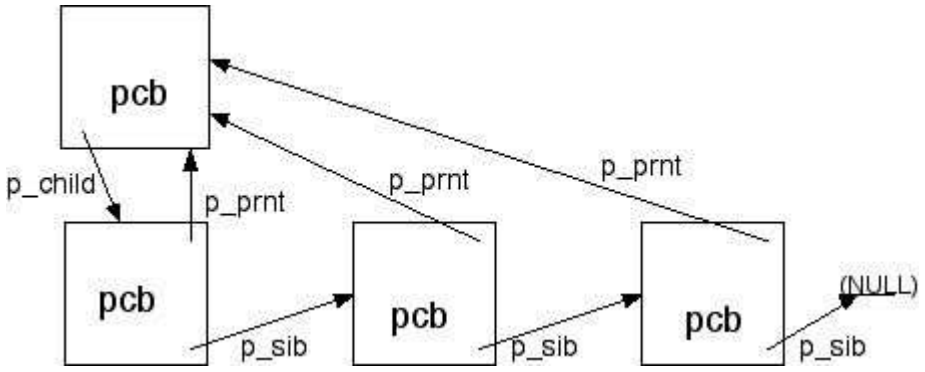

Figure 2.2: Process Tree

To support process trees there should be the following externally visible functions:

```
int emptyChild(pcb_t *p)
```

> /* Return TRUE if the ProcBlk pointed to by p has no children. Return FALSE otherwise. */

```
void insertChild(pcb_t *prnt, pcb_t *p)
```

> /* Make the ProcBlk pointed to by p a child of the ProcBlk pointed to by prnt. */

```
pcb_t *removeChild(pcb_t *p)
```

> /* Make the first child of the ProcBlk pointed to by p no longer a child of p. Return NULL if initially there were no children of p. Otherwise, return a pointer to this removed first child ProcBlk. */

```
pcb_t *outChild(pcb_t *p)
```

> /* Make the ProcBlk pointed to by p no longer the child of its parent. If the ProcBlk pointed to by p has no parent, return NULL; otherwise, return p. Note that the element pointed to by p need not be the first child of its parent. */

2.4 The Active Semaphore List

A *semaphore* is an important operating system concept which is needed for Phase 2/Level 3. While understanding semaphores is not needed for this level, this level nevertheless implements an important data structure/abstraction which supports JaeOS's implementation of semaphores.

For the purposes of this level it is sufficient to think of a semaphore as an integer. Associated with this integer is its address (semaphores, like all integers, have a physical address), and a process queue. A semaphore is *active* if there is at least one ProcBlk on the process queue associated it. (i.e. The process queue is not empty: `emptyProcQ(s_procq)` is FALSE.)

The following implementation is suggested: Maintain a sorted NULL-terminated single linearly linked list (using the `s_next` field) of semaphore descriptors whose

head is pointed to by the variable `semd_h`. The list `semd_h` points to will represent the *Active Semaphore List* (ASL). Keep the ASL sorted in ascending order using the `s_semAdd` field as the sort key.

```
/* semaphore descriptor type */
typedef struct semd_t {
    struct semd_t *s_next;     /* next element on the ASL */
    int           *s_semAdd;   /* pointer to the semaphore*/
    pcb_t         *s_procQ;    /* tail pointer to a       */
                               /* process queue           */
} semd _t;
```

Maintain a second list of semaphore descriptors, the *semdFree* list, to hold the unused semaphore descriptors. This list, whose head is pointed to by the variable `semdFree_h`, is kept, like the pcbFree list, as a NULL-terminated single linearly linked list (using the `s_next` field).

The semaphore descriptors themselves should be declared, like the ProcBlk's, as a static array of size MAXPROC of type `semd_t`.

There is no reason to make the ASL doubly linked, though for greater ASL traversal efficiency one may opt to place a dummy node at either the head or both the head and tail of the ASL. (In this case the size of the static array will increase by either one or two.)

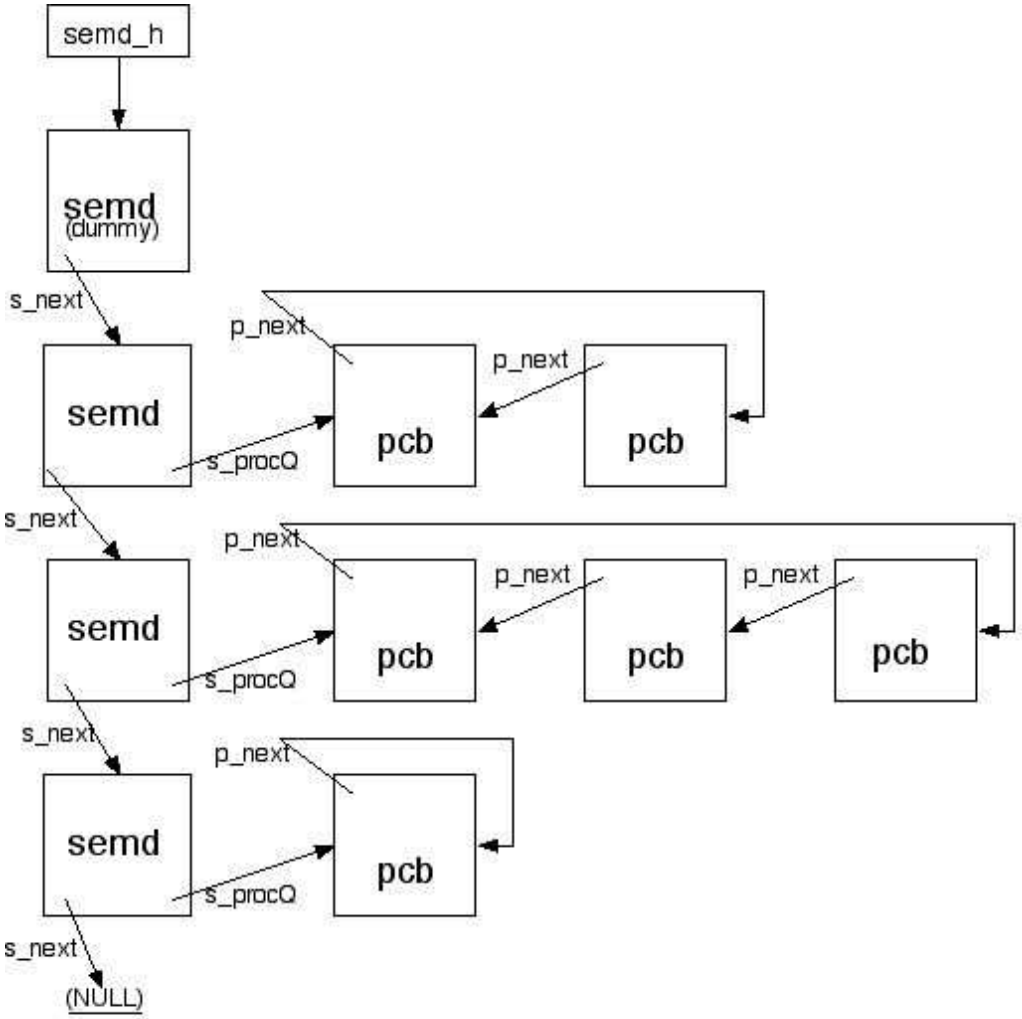

Figure 2.3: Active Semaphore List

To support the ASL there should be the following externally visible functions:

`int insertBlocked(int *semAdd, pcb_t *p)`

> /* Insert the ProcBlk pointed to by `p` at the tail of the process queue associated with the semaphore whose physical address is `semAdd` and set the semaphore address of `p` to semAdd. If the semaphore is currently not active (i.e. there is no descriptor for it in the ASL), allocate a new descriptor from the semdFree list, insert it in the ASL (at

the appropriate position), initialize all of the fields (i.e. set `s_semAdd` to `semAdd`, and `s_procq` to `mkEmptyProcQ()`), and proceed as above. If a new semaphore descriptor needs to be allocated and the semdFree list is empty, return TRUE. In all other cases return FALSE. */

`pcb_t *removeBlocked(int *semAdd)`

/* Search the ASL for a descriptor of this semaphore. If none is found, return NULL; otherwise, remove the first (i.e. head) ProcBlk from the process queue of the found semaphore descriptor and return a pointer to it. If the process queue for this semaphore becomes empty (`emptyProcQ(s_procq)` is TRUE), remove the semaphore descriptor from the ASL and return it to the semdFree list. */

`pcb_t *outBlocked(pcb_t *p)`

/* Remove the ProcBlk pointed to by `p` from the process queue associated with `p`'s semaphore (`p`$\rightarrow$ `p_semAdd`) on the ASL. If ProcBlk pointed to by `p` does not appear in the process queue associated with `p`'s semaphore, which is an error condition, return NULL; otherwise, return `p`. */

`pcb_t *headBlocked(int *semAdd)`

/* Return a pointer to the ProcBlk that is at the head of the process queue associated with the semaphore `semAdd`. Return NULL if `semAdd` is not found on the ASL or if the process queue associated with `semAdd` is empty. */

`void initASL(`

/* Initialize the semdFree list to contain all the elements of the array
static `semd_t semdTable[MAXPROC]`
This method will be only called once during data structure initialization. */

2.5 Nuts and Bolts

There is no one right way to implement the functionality of this level. One approach is to create two modules: one for the ASL and one for ProcBlk initialization/allocation/deallocation, process queue maintenance, and process tree maintenance.

The first module, `pcb.c`, in addition to the public and HIDDEN/private helper functions, will also contain the declaration for the private global variable that points to the head of the pcbFree list

```
HIDDEN pcb_t *pcbFree_h;
```

The ASL module, `asl.c`, in addition to the public and HIDDEN/private helper functions, will also contain the declarations for `semd_h` and `semdFree_h`

```
HIDDEN semd_t *semd_h, *semdFree_h;
```

Since the ASL module will make calls to the process queue module to manipulate the process queue associated with each active semaphore, this module should

```
#include ''pcb.e''
```

This will insure that the ASL can only use the externally visible functions from `pcb.c` for maintaining its process queues.

Furthermore, the declaration for `pcb_t` would then be placed in `types.h`. This is because many other modules will need to access this definition. The declaration for `semd_t` is placed in `asl.c` because no other module will ever need to access this definition; hence it is local to the module.

As with any non-trivial system, you are strongly encouraged to use the *make* program to maintain your code. A sample make file has been supplied for you to use.

2.6 Testing

There is a provided test file, `p1test.c` that will "exercise" your code.

You should compile the three source files separately using the commands:

```
arm-none-eabi-gcc -mcpu=arm7tdmi -c pcb.c
arm-none-eabi-gcc -mcpu=arm7tdmi -c asl.c
arm-none-eabi-gcc -mcpu=arm7tdmi -c p1test.c
```

The three object files 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
    p1test.o asl.o pcb.o /usr/include/uarm/crtso.o
    /usr/include/uarm/libuarm.o
```

The files `elf32ltsmip.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 two support files must be in their respective positions.

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

```
uarm
```

without any parameters loads the file `kernel.core.uarm` by default. See uarm for details on using the μARM simulator and it GUI interface.

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. These messages are also added to one of two memory buffers; `errbuf` for error messages and `okbuf` for all other messages. 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.

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) of JaeOS builds on previous levels in two key ways:

- Building on the exception (which includes device interrupts) handling facility of Level 1 (the ROM-Excpt handler is described in 2.4 - pops), 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), SuperVisor/Breakpoint calls (SVC/BKPT), TLB Management (TLB), and Interrupts (Ints).

- Using the data structures from Level 2 (Chapter 2), and the facility to handle both SVC/BKPT exceptions and Interrupts, 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 SVC/BKPT exceptions to the next level; the VM-I/O support level Level (see Chapter **??**).

Since virtual memory is not supported by JaeOS until Level 4, all addresses at this level are assumed to by 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.R1.M**=1) when it is either interrupted or executes the **svc** instruction, then **CP15.R1.M** 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 have an opportunity to execute. For simplicity's 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 (tail) pointer to a queue of ProcBlk's 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 Soft-block 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. μARM supports a **WAIT** instruction expressly for this purpose. See Section 5.1 - pops for more information about the **WAIT** instruction.

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. Populating the four New Areas in the Kernel Reserved Frame. (See 3.1 - pops.) For each of these *new* processor states:
   - 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.
   - Set the **CP15.R1** register to turn virtual memory (address translation) off.
2. Initialize the Level 2 (phase 1 - see Chapter 2) data structures:

   ```
   initPcbs()
   initSemd()
   ```

3. Initialize all nucleus maintained variables: Process Count, Soft-block Count, Ready Queue, and Current Process, etc.
4. Initialize all nucleus maintained semaphores (integers). 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 FIX LOS-TERMINALS), 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, 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

```
... = (unsigned int)test
```

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

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

6. Initialize/Set the psuedo-clock timer.

7. Call the scheduler.

Once `main()` calls the scheduler its task is completed since control will never return to `main()`. At this point the only mechanism for re-entering the nucleus is through an exception or an interrupt. 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 Kernel Reserved Frame, as defined in 3.1 - pops. Furthermore, the processor will be in kernel-mode with virtual memory disabled, all interrupts masked, and the Interval Timer disabled. 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 SVC/BKPT Exception Handling

A SuperVisor or Breakpoint exception occurs when an **SVC** or **BKPT** assembler instruction is executed. Assuming that the SVC/BKPT New Area in the Kernel Reserved Frame was correctly initialized during nucleus initialization, execution continues with the nucleus's SVC/BKPT exception handler.

A **SVC** exception is distinguished from a **BKPT** exception by the contents of **CP15**. **Cause** in the SVC/BKPT Old Area. SYSCALL exceptions are recognized via an exception code of *Sys* (4) while Breakpoint exceptions are recognized via an exception code of *BpE* (9).

By convention the executing process places appropriate values in user registers **A1**– **A4** immediately prior to executing a **SVC** or **BKPT** instruction. The nucleus will then perform some service on behalf of the process executing the **SVC** or **BKPT** instruction depending on the value found in **A1**.

In particular, if the process making a **SVC** request was in kernel-mode and **A1** contained a value in the range [1..8] then the nucleus should perform one of the services described below.

3.3.1 Create_Process (SVC 1)

When requested, this service causes a new process, said to be a *progeny* of the caller, to be created. **A3** 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. 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 value 0 in the caller's **A1** .

The SVC 1 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 the **SVC** 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 (SVC 2)

This services causes the executing process to cease to exist. In addition, recursively, all progeny of this process are terminated as well. Execution of this instruction does not complete until *all* progeny are terminated.

The SVC 2 service is requested by the calling process by placing the value 2 in **A1** and then executing the **SVC** instruction.

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

```
void SYSCALL (TERMINATEPROCESS)
```

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

3.3.3 Verhogen (V) (SVC 3)

When this service is requested, it is interpreted by the nucleus as a request to perform a **V** operation on a semaphore.

The **V** or SVC 3 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**, and then executing the **SVC** instruction.

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

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

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

3.3.4 Passeren (P) (SVC 4)

When this services is requested, it is interpreted by the nucleus as a request to perform a **P** operation on a semaphore.

The **P** or SVC 4 service is requested by the calling process by placing the value 4 in **A1**, the physical address of the semaphore to be P'ed in **A2**, and then executing the **SVC** instruction.

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

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

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

3.3.5 Specify_Exception_State_Vector (SVC 5)

When this service is requested, three pieces of information need to be supplied to the nucleus:

- The type of exception for which an *Exception State Vector* is being established. This information should be in **A2** and will represent:

  0: TLB exceptions

  1: PgmTrap exceptions

  2: SVC/BKPT exceptions

- The address into which the old processor state is to be stored when an exception occurs while running this process. This address, the *XXX Old Area Address*, where XXX is either TLB, PgmTrap, or SVC/BKPT, should be **A3**.

- The processor state area that is to be taken as the new processor state if an exception occurs while running this process. This address, the *XXX New Area Address*, where XXX is either TLB, PgmTrap, or SVC/BKPT, should be in **A4**.

The nucleus, when this service is requested, will save the contents of **A3** and **A4** (in the invoking process'es ProcBlk) to facilitate "passing up" handling of the respective exception when (and if) one occurs while this process is executing. When an exception occurs for which an Exception State Vector has been specified for, the nucleus stores the processor state at the time of the exception in the area pointed to by the address in **A3**, and loads the new processor state from the area pointed to by the address given in **A4**.

Each process may request a SVC 5 service **at most** once for each of the three exception types. An attempt to request a SVC 5 service more than once per exception type by any process should be construed as an error and treated as a SVC 2.

If an exception occurs while running a process which has not specified an Exception State Vector for that exception type, then the nucleus should treat the exception as a SVC 2 as well.

The SVC 5 service is requested by the calling process by placing the value 5 in **A1**, an exception code ([0..2]) in **A2**, physical addresses of processor state areas in **A3** and **A4**, and then executing the **SVC** instruction.

The following C code can be used to request a SVC 5:

```
   void SYSCALL (SPECTRAPVEC, int type, state_t *oldp,
state_t *newp)
```

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

3.3.6 Get_CPU_Time (SVC 6)

When this service is requested, it causes the processor time (in microseconds) used by the requesting process to be placed/returned in the caller's **A1**. This means that the nucleus must record (in the ProcBlk) the amount of processor time used by each process.

The SVC6 service is requested by the calling process by placing the value 6 in **A1** and then executing the **SVC** instruction.

The following C code can be used to request a SVC 6:

```
   cpu_t SYSCALL (GETCPUTIME)
```

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

3.3.7 Wait_For_Clock (SVC 7)

This instruction performs a **P** operation on the nucleus maintained pseudo-clock timer semaphore. This semaphore is **V**'ed every 100 milliseconds automatically

by the nucleus.

The SVC 7 service is requested by the calling process by placing the value 7 in **A1** and then executing the **SVC** instruction.

The following C code can be used to request a SVC 7:

```
void SYSCALL (WAITCLOCK)
```

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

3.3.8 Wait_for_IO_Device (SVC 8)

This service performs a **P** operation on the semaphore that the nucleus maintains for the I/O device indicated by the values in **A2**, **A3**, and optionally **A4**.

Note that terminal devices are two independent sub-devices (see FIX LOSTERMINALS), and are handled by the SVC 8 service as two independent devices. Hence each terminal device has two nucleus maintained semaphores for it; one for character receipt and one for character transmission.

As discussed in Section 3.6 the nucleus will perform a **V** operation on the nucleus maintained semaphore whenever that (sub)device generates an interrupt.

Once the process resumes after the occurrence of the anticipated interrupt, the (sub)device's status word is returned in **A1**. For character transmission and receipt, the status word, in addition to containing a device completion code, will also contain the character transmitted or received.

As described below in Section 3.6 it is possible that the interrupt can occur prior to the request for the SVC 8 service. In this case the requesting process will not block as a result of the **P** operation and the interrupting device's status word, which was stored off, can now be placed in **A1** prior to resuming execution.

The SVC 8 service is requested by the calling process by placing the value 8 in **A1**, the interrupt line number in **A2**, the device number in **A3** ([0. . .7]) , TRUE or FALSE in **A4** to indicate if waiting for a terminal read operation, and then executing a **SVC** instruction.

The following C code can be used to request a SVC 8:

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

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

3.3.9 SVC 1 – SVC 8 in User-Mode

The above eight 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 SVC/BKPT Old Area to the PgmTrap Old Area, setting **CP15**. **Cause**.**ExcCode** in the PgmTrap Old Area to *RI* (Reserved Instruction), and calling JaeOS's PgmTrap exception handler.

3.3.10 Breakpoint Exceptions and SVC 9 and Above Exceptions

The nucleus will directly handle all SVC 1 – SVC 8 requests. The ROM-Excpt handler will also directly handle some Breakpoint exceptions; those where the requesting process was executing in kernel-mode and **A1** contained the code for either **WAIT**, **FORK**, **PANIC**, or **HALT**.

For all other SuperVisor and Breakpoint exceptions the nucleus's SVC/BKPT exception handler will take one of two actions depending on whether the offending (i.e. Current) process has performed a SVC 5 for SVC/BKPT exceptions:

- If the offending process has NOT issued a SVC 5 for SVC/BKPT exceptions, then the SVC/BKPT exception should be handled like a SVC 2: the current process and all its progeny are terminated.

- If the offending process has issued a SVC 5 for SVC/BKPT exceptions, the handling of the SVC/BKPT exception is "passed up." The processor state is moved from the SVC/BKPT Old Area into the processor state area whose address was recorded in the ProcBlk as the SVC/BKPT Old Area Address. Finally, the processor state whose address was recorded in the ProcBlk as the SVC/BKPT New Area Address 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 described in 2.5 - pops. Assuming that the PgmTrap New Area in the Kernel 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 SVC 5 for PgmTrap exceptions:

- If the offending process has NOT issued a SVC 5 for PgmTrap exceptions, then the PgmTrap exception should be handled like a SVC 2: the current process and all its progeny are terminated.

- If the offending process has issued a SVC 5 for PgmTrap exceptions, the handling of the PgmTrap is "passed up." The processor state is moved from the PgmTrap Old Area into the processor state area whose address was recorded in the ProcBlk as the PgmTrap Old Area Address. Finally, the processor state whose address was recorded in the ProcBlk as the PgmTrap New Area Address is made the current processor state.

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. All the various ways a failure can occur are described in 2.2 - pops. Assuming that the TLB New Area in the Kernel 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 SVC 5 for TLB exceptions:

- If the offending process has NOT issued a SVC 5 for TLB exceptions, then the TLB exception should be handled like a SVC 2: the current process and all its progeny are terminated.

- If the offending process has issued a SVC 5 for TLB exceptions, the handling of the PgmTrap is "passed up." The processor state is moved from the TLB Old Area into the processor state area whose address was recorded in the ProcBlk as the TLB Old Area Address. Finally, the processor state

whose address was recorded in the ProcBlk as the TLB New Area Address is made the current processor state.

3.6 Interrupt Exception Handling

A device interrupt occurs when a previously initiated I/O request completes or when the Interval Timer makes a 0x0000.0000 $\Rightarrow$ 0xFFFF.FFFF transition. Assuming that the Ints New Area in the Kernel Reserved Frame was correctly initialized during nucleus initialization, then after the processor's and ROM-Excpt handler's actions when an Ints exception is raised, execution continues with the nucleus's Ints exception handler. Which interrupt lines have pending interrupts is set in **CP15**. **Cause**. (See 2.2 - pops.) Furthermore, for interrupt lines 3–7 the Interrupting Devices Bit Map (see 3.1 - pops) will indicate which devices on each of these interrupt lines have a pending interrupt. Since JaeOS is intended for uniprocessor environments only, interrupt line 0 may safely be ignored.

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 FIX LOS-TERMINALS, 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 Interval Timer (interrupt line 1) 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 timer device 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.

An interrupt for the Interval Timer is acknowledged by loading the timer with a new value.

- Perform a **V** operation on the nucleus maintained semaphore associated with the interrupting (sub)device. The nucleus maintains two semaphores for each terminal sub-device. For Interval Timer interrupts that represent a pseudo-clock tick (see Section 3.7.1), perform the **V** operation on the nucleus maintained pseudo-clock timer semaphore.

- If the SVC 8 for this interrupt was requested prior to the handling of this interrupt, recognized by the **V** operation above unblocking a blocked process, store the interrupting (sub)device's status word in the newly unblocked process'es **A2**. If the SVC 8 for this interrupt has not yet been requested, recognized by the V operation not unblocking any process, store off the interrupting device's status word until the SVC 8 is eventually requested.

3.7 Nuts and Bolts

3.7.1 Timing Issues

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

- Generate an interrupt to signal the end of processes' time slices.

- 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 (SVC 4, SVC 7, or SVC 8) or terminate (SVC 2) long 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 SVC 7, the pseudo-clock timer semaphore, if left unadjusted, will grow by 1 every 100 milliseconds. This means that if a process,

after 500 milliseconds requests a SVC 7, and there were no intervening SVC 7 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. SVC 6). 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 timer device is mechanisms for implementing JaeOS's scheduling 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 SuperVisor request needs to be measured for pseudo-clock tick purposes, which process, if any, should be "charged" with this time.

It is important to understand the functional differences between the two μARM timer devices. This includes, but is not limited to understanding that the TOD clock counts up while the Interval timer counts down, and the TOD clock cannot be changed.

3.7.2 Returning from a SVC/BKPT Exception

SuperVisor calls that do not result in process termination return control to the requesting process'es execution stream. This is done either immediately (e.g. SVC 6) or after the process is blocked and eventually unblocked (e.g. SVC 8). 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 **SVC** assembly instruction.

Without intervention, returning control to the requesting process will result in an infinite loop of SuperVisor calls. To avoid this the **pc** must be incremented by 4 (i.e. the μARM wordsize) prior to returning control to the interrupted execution stream.

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. (As described in 5.1 - pops, **LDST** is a ROM-based service/instruction implemented using a Breakpoint exception.)

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 is an invariant that the absolute value of the semaphore equal the number of ProcBlk'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 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 the ready queue, blocked on a device semaphore, or blocked on a non-device semaphore.

3.7.5 Module Decomposition

One possible module decomposition is as follows:

1. `initial.c` This module implements `main()` and exports the nucleus's global variables. (e.g. Process Count, device semaphores, etc.)

2. `interrupts.c` This module implements the device interrupt exception handler. This module will process all the device interrupts, including Interval Timer interrupts, converting device interrupts into **V** operations on the appropriate semaphores.

3. `exceptions.c` This module implements the TLB PgmTrap and SVC/BKPT exception handlers.

4. `scheduler.c` This module implements JaeOS's process scheduler and deadlock detector.

3.7.6 Accessing the `libumps` Library

As described in 5.1 - pops, 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.

3.7.7 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.o
```

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

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

```
uarm
```

without any parameters loads the file `kernel.core.umps` by default. See FIX LOSGUIfor details on using the μARM simulator and it GUI interface.

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.

Bibliography

[1] ALVISI, L., AND SCHNEIDER, F. A graphical interface for CHIP. Tech. rep., Cornell University, 1996. Technical Report TR 96-1587.

[2] BABAOGLU, O., BUSSAN, M., DRUMMOND, R., AND SCHNEIDER, F. Documentation for the CHIP computer system, 1988.

[3] BABAOGLU, O., AND SCHNEIDER, F. The HOCA operating system specifications, 1990.

[4] DIJKSTRA, E. The structure of the THE-multiprogramming system. *Commun. ACM 11*, 3 (may 1968).

[5] GOLDWEBER, M., DAVOLI, R., AND JONJIC, T. Supporting operating systems projects using the μ mps2 hardware simulator. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education* (New York, NY, USA, 2012), ITiCSE '12, ACM, pp. 63–68.

[6] GOLDWEBER, M., DAVOLI, R., AND MORSIANI, M. The kaya OS project and the μmps hardware emulator. In *Proceedings of the 10th Annual Conference on Innovation and Technology in Computer Science Education ITiCSE '05* (2005).

[7] MORSIANI, M. ICARO.S resource page. http://www.cs.unibo.it/mps/icaros.html.

[8] MORSIANI, M. MPS resource page. http://www.cs.unibo.it/mps.

[9] MORSIANI, M., AND DAVOLI, R. Learning operating systems structure and implementation through the MPS computer system simulator. In *Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education* (1999).