Student Guide to the Kaya Operating System Project

The Virtual Square Lab

Michael Goldweber
Xavier University

Renzo Davoli
University of Bologna

Kaya, μMPS & μMPS2 are products of the Virtual Square Lab.
See `virtualsquare.org/` and `wiki.virtualsquare.org/`.
The μMPS & μMPS2 home page is `www.cs.xu.edu/uMPS/`

Kaya v1.2

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, but 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!

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. This new 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. Finally in 2011 μMPS was updated by Tomislav Jonjic to μMPS2 with a new GUI and multiprocessor support. μMPS2 is 100% backward compatible with μMPS. While μMPS is still available (though unsupported), this guide assumes the use of μMPS2. While the Kaya project described herein can be implemented on μMPS, this version (v1.2) makes use of a couple of hardware features new to μMPS2.

A raw machine emulator, such as μMPS2, which is fully described in "μMPS2 Principles of Operation," can support a wide variety of undergraduate, and graduate-level projects. The Kaya project is just one such project. The Virtual Square Lab, which produced both Kaya and μMPS2 is also currently producing additional projects for μMPS2 as well as for VDE, the Virtual Distributed Ethernet tool also produced by the Virtual Square Lab.

These other projects, like Kaya, are all designed to be accomplished by advanced undergraduates working either solo or in teams in the context of a semester-long multi-phase project.

Finally Renzo and myself wish to offer our heartfelt thanks to Mauro Morsiani who generously donated his time to modify MPS into μMPS, and Tomislav Jonjic who extended μMPS to create the backward compatible μMPS2. We also wish to 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, 2009
Updated: August, 2011

The least of learning is done in the classrooms.

Thomas Merton

1 Introduction

The Kaya OS 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.

Kaya 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[6] and ICAROS[5], for implementation on the Microprocessor (without) Pipeline Stages (MPS)[7, 6].

Level 0: The base hardware of μMPS2.
There are two versions of the μMPS hardware; the original μMPS emulator, and μMPS2, a 100% backwards compatible extension of μMPS with an improved GUI and multiprocessor support. While both emulators are still available (though the original μMPS is no longer supported), this guide assumes the use of μMPS2 due to its superior GUI and additional hardware features.

Level 1: The additional services provided in ROM. This includes the services pro-

vided 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 (i.e. **LDST**, **FORK**, **PANIC**, and **HALT**).

The μMPS2 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 4). 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 μMPS2 Principles of Operation will have a *pops* suffix. e.g. A reference to the chapter on Exception Handling will be denoted: Chapter 3-*pops*.

  As there are two versions of the μMPS emulator, there are two versions of the Principles of Operation manual. As the original μMPS emulator is no longer supported, this guide assumes the use of μMPS2 Principles of Operation.

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 Kaya 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 Kaya supports no more that *MAXPROC* concurrent processes; where MAXPROC should be set to 20 (in the file CONST.H).[1] 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

[1]The recommended installation location for a "starter" CONST.H is /USR/LOCAL/SHARE/KAYA/

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 value 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:
  `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.)

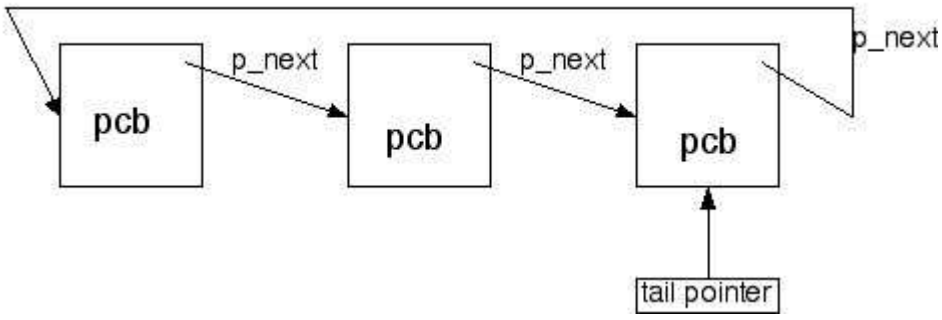

Figure 2.1: Process Queue

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. */

```
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.

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. */

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

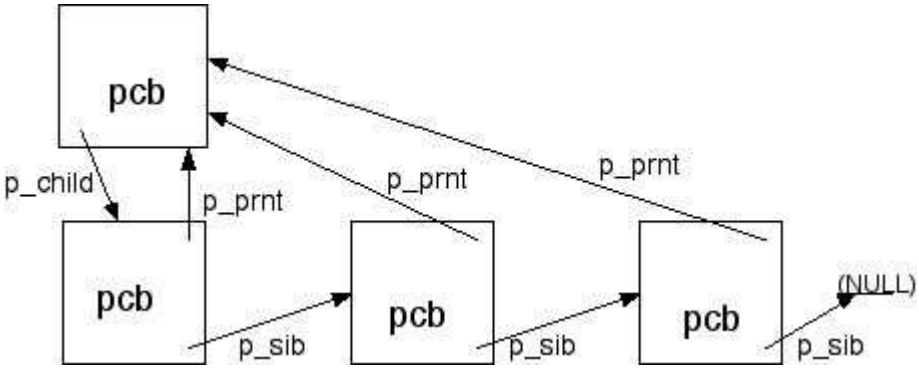

Figure 2.2: Process Tree

/* 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 Kaya'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.)

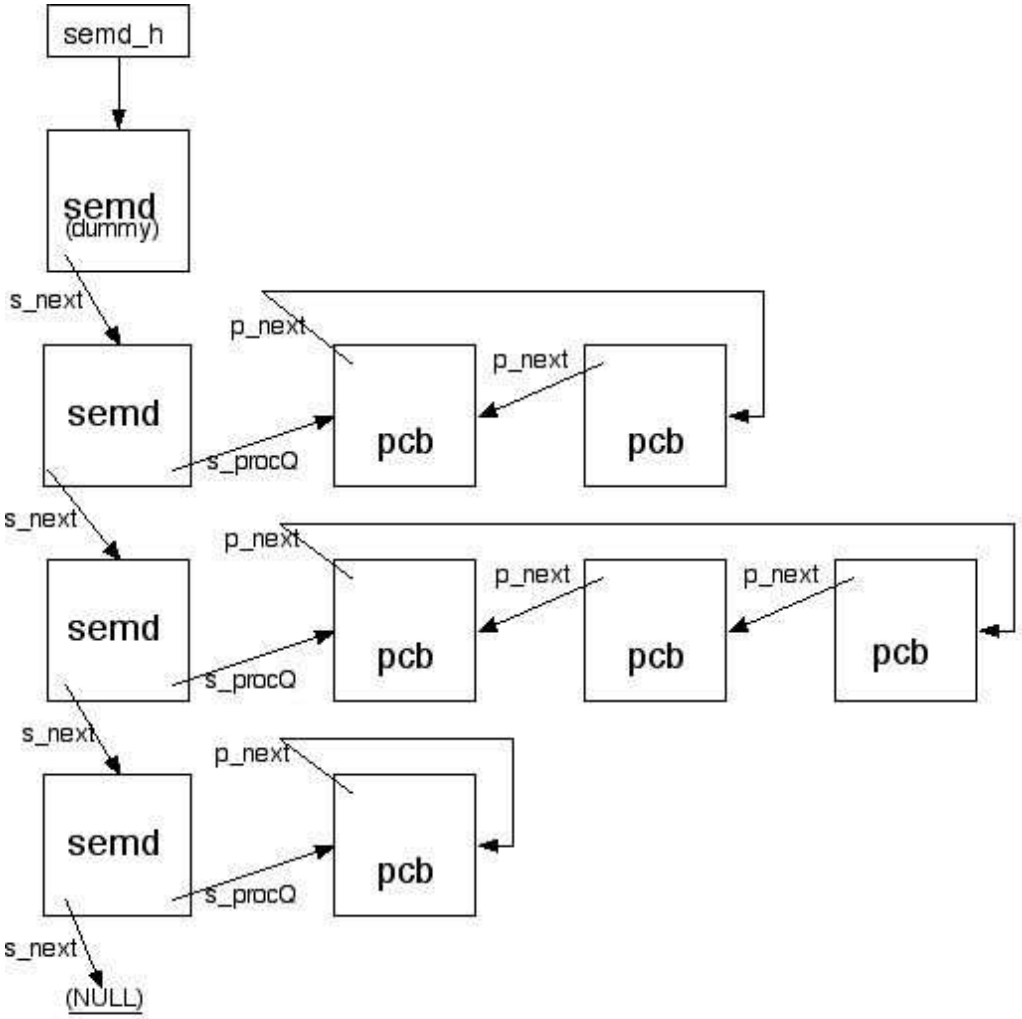

Figure 2.3: Active Semaphore List

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. */

`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 second 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.[2] 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.[3]

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

```
MIPSEL-LINUX-GCC -ANSI -PEDANTIC -WALL -C PCB.C
MIPSEL-LINUX-GCC -ANSI -PEDANTIC -WALL -C ASL.C
MIPSEL-LINUX-GCC -ANSI -PEDANTIC -WALL -C P1TEST.C
```

The three object files should then be linked together using the command:

```
MIPSEL-LINUX-LD -T
      /USR/LOCAL/SHARE/UMPS2/ELF32LTSMIP.H.UMPSCORE.X
      /USR/LOCAL/LIB/UMPS2/CRTSO.O P1TEST.O ASL.O PCB.O
      /USR/LOCAL/LIB/UMPS2/LIBUMPS.O -O KERNEL
```

[2] The recommended installation location for a "starter" TYPES.H is /USR/LOCAL/SHARE/KAYA/

[3] The recommended installation location for P1TEST.C along with a sample Makefile is /USR/LOCAL/SHARE/KAYA/

The files ELF32LTSMIP.H.UMPSCORE.X, CRTSO.O, and LIBUMPS.O are part of the μMPS2 distribution. /USR/LOCAL/XXXX/UMPS2/ 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.[4]

The linker produces a file in the ELF object file format which needs to be converted prior to its use with μMPS2. This is done with the command:

```
UMPS2-ELF2UMPS -K KERNEL
```

which produces the file KERNEL.CORE.UMPS

Finally, your code can be tested by launching μMPS2. Entering:

```
UMPS2
```

without any parameters loads the file KERNEL.CORE.UMPS by default. See Chapter 9-*pops* for details on using the μMPS2 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 μMPS2 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, μMPS2 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.

[4] As documented in Section 8.1-*pops*, if one is working on a big-endian machine one should modify the above commands appropriately; substitute MIPS- for MIPSEL- above.

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 Kaya builds on previous levels in two key ways:

- Building on the exception handling facility of Level 1 (the ROM-Excpt handler is described in Chapter 3-*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), SYSCALL/Breakpoint (SYS/Bp), TLB Management (TLB), and Interrupts (Ints).

- Using the data structures from Level 2 (Chapter 2), and the facility to handle both SYS/Bp exceptions and Interrupts –timer interrupts in particular– provide a process scheduler.

The purpose of the nucleus is to provide an environment in which asynchronous sequential processes (i.e. heavyweight threads) exist, each making forward progress as they take turns sharing the processor. Furthermore, the nucleus provides these processes with exception handling routines, low-level synchronization primitives, and a facility for "passing up" the handling of PgmTrap, TLB exceptions and certain SYS/Bp exceptions to the next level; the VM-I/O support level Level (see Chapter 4).

Since virtual memory is not supported by Kaya 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 (**Status.VMc**=1) when it is either interrupted or executes a **SYSCALL**, then **Status.VMc** 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 Kaya 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. μMPS2 supports a **WAIT** instruction expressly for this purpose. See Section Section 6.2-*pops* for more information about the **WAIT** instruction.

3.2 Nucleus Initialization

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

1. Populate the four New Areas in the ROM Reserved Frame. (See Section 3.2.2-*pops*.) For each New processor state:
   - Set the **PC** to the address of your nucleus function that is to handle exceptions of that type.
   - Set the **$SP** to RAMTOP. Each exception handler will use the last frame of RAM for its stack.
   - Set the **Status** register to mask all interrupts, turn virtual memory off, enable the processor Local Timer, and be in kernel-mode.
2. Initialize the Level 2 (phase 1 - see Chapter 2) data structures:
   ```
   initPcbs()
   initSemd()
   ```
3. Initialize all nucleus maintained variables: Process Count, Soft-block Count, Ready Queue, and Current Process.
4. Initialize all nucleus maintained semaphores. In addition to the above nucleus variables, there is one semaphore variable for each external (sub)device in μMPS2, plus a semaphore to represent a pseudo-clock timer. Since terminal devices are actually two independent sub-devices (see Section 5.7-*pops*), the nucleus maintains two semaphores for each terminal device. All of these semaphores need to be initialized to zero.
5. Instantiate a single process and place its ProcBlk in the Ready Queue. A process is instantiated by allocating a ProcBlk (i.e. `allocPcb()`), and initializing the processor state that is part of the ProcBlk. In particular this process needs to have interrupts enabled, virtual memory off, the processor Local Timer enabled, kernel-mode on, **$SP** set to RAMTOP-FRAMESIZE

(i.e. use the penultimate RAM frame for its stack), and its **PC** set to the address of `test`. Test is a supplied function/process that will help you debug your nucleus. One can assign a variable (i.e. the **PC**) the address of a function by using

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

where `memaddr`, in TYPES.H has been aliased to `unsigned int`.

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

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

For rather technical reasons that are somewhat explained in Section 8.2-*pops*, 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.) Hence this will be done when initializing the four New Areas as well as the processor state that defines this single process.

6. Call the scheduler.

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

At boot/reset time the nucleus is loaded into RAM beginning with the second frame of RAM; 0x2000.1000. The first frame of RAM is the ROM Reserved Frame, as defined in Section 3.2.2-*pops*. Furthermore, the processor will be in kernel-mode with virtual memory disabled, all interrupts masked, and the processor Local Timer disabled. The **PC** is assigned 0x2000.1000 and the **$SP**, which was initially set to RAMTOP at boot-time, will now be some value less than RAMTOP due to the activation record for `main()` that now sits on the stack.

3.3 SYS/Bp Exception Handling

A SYSCALL or Breakpoint exception occurs when a **SYSCALL** or **BREAK** assembler instruction is executed. Assuming that the SYS/Bp New Area in the ROM Reserved Frame was correctly initialized during nucleus initialization, then after

the processor's and ROM-Excpt handler's actions when a SYSCALL or Breakpoint exception is raised, execution continues with the nucleus's SYS/Bp exception handler.

A SYSCALL exception is distinguished from a Breakpoint exception by the contents of **Cause.ExcCode** in the SYS/Bp Old Area. SYSCALL exceptions are recognized via an exception code of *Sys* (8) while Breakpoint exceptions are recognized via an exception code of *Bp* (9).

By convention the executing process places appropriate values in user registers **a0**– **a3** immediately prior to executing a **SYSCALL** or **BREAK** instruction. The nucleus will then perform some service on behalf of the process executing the **SYSCALL** or **BREAK** instruction depending on the value found in **a0**.

In particular, if the process making a **SYSCALL** request was in kernel-mode and **a0** contained a value in the range [1..8] then the nucleus should perform one of the services described below.

3.3.1 Create_Process (SYS1)

When requested, this service causes a new process, said to be a *progeny* of the caller, to be created. **a1** 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 **v0**, otherwise, return the value 0 in the caller's **v0**.

The SYS1 service is requested by the calling process by placing the value 1 in **a0**, the physical address of a processor state in **a1**, and then executing a **SYSCALL** instruction.

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

```
int SYSCALL (CREATEPROCESS, state_t *statep)
```

Where the mnemonic constant `CREATEPROCESS` has the value of 1.

3.3.2 Terminate_Process (SYS2)

This services causes the executing process 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 SYS2 service is requested by the calling process by placing the value 2 in **a0** and then executing a **SYSCALL** 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) (SYS3)

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 SYS3 service is requested by the calling process by placing the value 3 in **a0**, the physical address of the semaphore to be V'ed in **a1**, and then executing a **SYSCALL** 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) (SYS4)

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 SYS4 service is requested by the calling process by placing the value 4 in **a0**, the physical address of the semaphore to be P'ed in **a1**, and then executing a **SYSCALL** 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 (SYS5)

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 **a1** and will represent:

  0: TLB exceptions

  1: PgmTrap exceptions

  2: SYS/Bp 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 SYS/Bp, should be **a2**.

- 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 SYS/Bp, should be in **a3**.

The nucleus, when this service is requested, will save the contents of **a2** and **a3** (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 **a2**, and loads the new processor state from the area pointed to by the address given in **a3**.

Each process may request a SYS5 service **at most** once for each of the three exception types. An attempt to request a SYS5 service more than once per exception type by any process should be construed as an error and treated as a SYS2.

If an exception occurs while running a process which has not specified an Exception State Vector for that exception type, then the nucleus should treat the exception as a SYS2 as well.

The SYS5 service is requested by the calling process by placing the value 5 in **a0**, an exception code ([0..2]) in **a1**, physical addresses of processor state areas in **a2** and **a3**, and then executing a **SYSCALL** instruction.

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

```
   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 (SYS6)

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 **v0**. This means that the nucleus must record (in the ProcBlk) the amount of processor time used by each process.

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

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

```
cpu_t SYSCALL (GETCPUTIME)
```

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

3.3.7 Wait_For_Clock (SYS7)

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

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

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

```
void SYSCALL (WAITCLOCK)
```

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

3.3.8 Wait_for_IO_Device (SYS8)

This service performs a P operation on the semaphore that the nucleus maintains for the I/O device indicated by the values in **a1**, **a2**, and optionally **a3**.

Note that terminal devices are two independent sub-devices (see Section 5.7-*pops*), and are handled by the SYS8 service as two independent devices. Hence each terminal device has two nucleus maintained semaphores for it; one for character receipt and one for character transmission.

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

Once the process resumes after the occurrence of the anticipated interrupt, the (sub)device's status word is returned in **v0**. For character transmission and receipt, the status word, in addition to containing a device completion code, will also contain the character transmitted or received.

As described below in Section 3.6 it is possible that the interrupt can occur prior to the request for the SYS8 service. In this case the requesting process will not block as a result of the P operation and the interrupting device's status word, which was stored off, can now be placed in **v0** prior to resuming execution.

The SYS8 service is requested by the calling process by placing the value 8 in **a0**, the interrupt line number in **a1**, the device number in **a2** ([0. . .7]) , TRUE or FALSE in **a3** to indicate if waiting for a terminal read operation, and then executing a **SYSCALL** instruction.

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

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

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

3.3.9 SYS1-SYS8 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 Kaya should simulate a PgmTrap exception when a privileged service is requested in user-mode. This is done by moving the processor state from the SYS/Bp Old Area to the PgmTrap Old Area, setting **Cause.ExcCode** in the PgmTrap Old Area to *RI* (Reserved Instruction), and calling Kaya's PgmTrap exception handler.

3.3.10 Breakpoint Exceptions and SYS9 and Above Exceptions

The nucleus will directly handle all SYS1-SYS8 requests. The ROM-Excpt handler will also directly handle some Breakpoint exceptions; those where the requesting process was executing in kernel-mode and **a0** contained the code for either **LDST**, **FORK**, **PANIC**, or **HALT**[1. . .4].

For all other SYSCALL and Breakpoint exceptions the nucleus's SYS/Bp exception handler will take one of two actions depending on whether the offending (i.e. Current) process has performed a SYS5 for SYS/Bp exceptions:

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

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

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 Section 3.1.1-*pops*. Assuming that the PgmTrap New Area in the ROM Reserved Frame was correctly initialized during nucleus initialization, then after the processor's and ROM-Excpt handler's actions when a PgmTrap exception is raised, execution continues with the nucleus's PgmTrap exception handler. The cause of the PgmTrap exception will be set in **Cause.ExcCode** in the PgmTrap Old Area.

The nucleus's PgmTrap exception handler will take one of two actions depending on whether the offending (i.e. Current) process has performed a SYS5 for PgmTrap exceptions:

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

- If the offending process has issued a SYS5 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 μMPS2 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 Section 3.1.3-*pops*. Assuming that the TLB New Area in the ROM Reserved Frame was correctly initialized during nucleus initialization, then after the processor's and ROM-Excpt handler's actions when a TLB exception is raised, execution continues with the nucleus's TLB exception handler. The cause of the TLB exception will be set in **Cause.ExcCode** in the TLB Old Area.

The nucleus's TLB exception handler will take one of two actions depending on whether the offending (i.e. Current) process has performed a SYS5 for TLB exceptions:

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

- If the offending process has issued a SYS5 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 either a previously initiated I/O request completes or when either a processor Local Timer or the Interval Timer makes a 0x0000.0000 $\Rightarrow$ 0xFFFF.FFFF transition. Assuming that the Ints New Area in the ROM Reserved Frame was correctly initialized during nucleus initialization, then after the processor's and ROM-Excpt handler's actions when an Ints exception is raised, execution continues with the nucleus's Ints exception handler. Which interrupt lines have pending interrupts is set in **Cause.IP**. (See Section 3.3-*pops*.) Furthermore, for interrupt lines 3–7 the Interrupting Devices Bit Map (see Section 5.2.4-*pops*) will indicate which devices on each of these interrupt lines have a pending interrupt. Since Kaya 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 Section 5.7-*pops*, terminal devices are actually two sub-devices; a transmitter and a receiver. These two sub-devices operate independently and concurrently. Both sub-devices may have an interrupt pending simultaneously.

For purposes of prioritizing pending interrupts, terminal transmission (i.e. writing to the terminal) is of higher priority than terminal receipt (i.e. reading from the terminal). Hence the processor Local 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 two timer devices (the system-wide Interval Timer, and the processor Local Timer) 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 a timer device 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 SYS8 for this interrupt was requested prior to the handling of this interrupt, recognized by the V operation above unblocking a blocked process, store the interrupting (sub)device's status word in the newly unblocked process'es **v0**. If the SYS8 for this interrupt has not yet been requested, recognized by the V operation not unblocking any process, store off the interrupting device's status word until the SYS8 is eventually requested.

3.7 Nuts and Bolts

3.7.1 Timing Issues

While μMPS2 has three clocks, the TOD clock, Interval Timer, and the processor Local Timer, only the Interval Timer and the processor Local Timer can generate interrupts. This fits nicely with the two primary 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.

Given the system-wide nature of the Interval Timer, it is recommended that this device be used for to keep track of pseudo-clock ticks. This leaves the processor Local Timer free for implementing processor scheduling.

When no process requests a SYS7, the pseudo-clock timer semaphore, if left unadjusted, will grow by 1 every 100 milliseconds. This means that if a process, after 500 milliseconds requests a SYS7, and there were no intervening SYS7 requests, it will not block until the next pseudo-clock tick as hoped for, but will immediately resume its execution stream. Therefore at each pseudo-clock tick, if no process was unblocked by the V operation (i.e. the semaphore's value after the increment performed during the V operation was greater than zero), the semaphore's value should be decremented by one (i.e. reset to zero).

The opposite is also true; if more than one process requests a SYS7 in between two adjacent pseudo-clock ticks then at the next pseudo-clock tick, all of the waiting processes should be unblocked and not just the process that was waiting the longest.

The 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 ProcBlk structure. While the processor Local 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 three timer devices are mechanisms for implementing Kaya'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 purposes, which process, if any, should be "charged" with this time.

It is important to understand the functional differences between the three μMPS2 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 processor Local Timer is implementation dependent (i.e. unspecified) when the processor Local Timer enable bit is off. (i.e. The timer may or may not continue to decrement when **Status.TE**=0.)

It is typical in a multiprocessor system for each processor to have its own local timer in addition to a system wide timer. μMPS2 is a multiprocessor emulator set to default to a processor count of one. Hence there are two available interrupt-generating timer devices. The original μMPS emulator was a strict uniprocessor system and therefore did not support a processor Local Timer. Prior to μMPS2, to accomplish Kaya's timing needs, one used the Interval Timer for both timing purposes; process scheduling and pseudo-clock ticks. It is an worthwhile exercise, for the interested student, to implement Kaya's timing needs ignoring the processor Local Timer and *overloading* the Interval Timer to generate both types of timing interrupts.

3.7.2 Returning from a SYS/Bp Exception

SYSCALL's that do not result in process termination return control to the requesting process'es execution stream. This is done either immediately (e.g. SYS6) or after the process is blocked and eventually unblocked (e.g. SYS8). In any event the **PC** that was saved is, 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 μMPS2 wordsize) prior to returning control to the interrupted execution stream. While the **PC** needs to be altered, do not, in this case, make a parallel assignment to **t9**.

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 Chapter 6-*pops*, **LDST** is a ROM-based service/instruction implemented using a Breakpoint exception.) Whenever the ROM-Excpt handler loads a processor state a pop operation is performed on the **KU/IE** and **VM** stacks. (See Section 6.2.1-*pops*.)

This has implications whenever one is setting the **Status** register's **VM**, **KU**, or **IE** bits. One must set the *previous* bits (i.e. **VMp**, **IEp** & **KUp**) and not the

current bits (i.e. **VMc**, **IEc** & **KUc**) for the desired assignment to take effect after the ROM-Excpt handler loads the processor state.

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 SYS/Bp exception handlers.

4. SCHEDULER.C This module implements Kaya's process scheduler and deadlock detector.

3.7.6 Accessing the `libumps` Library

As described in Chapter 6-*pops*, accessing the **CP0** registers and the ROM-implemented services/instructions in C is via the libumps library. Simply include the line

```
#include ''/usr/local/include/umps2/umps/libumps.e''
```

The file LIBUMPS.E is part of the μMPS2 distribution.
/USR/LOCAL/INCLUDE/UMPS2/UMPS/ is the recommended installation location for this file. 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.[1]
You should individually compile all the source files from both phase1 and phase2 in addition to the phase2 test file using the command:

```
MIPSEL-LINUX-GCC -ANSI -PEDANTIC -WALL -C FILENAME.C
```

All of the object files should then be linked together using the command:

```
MIPSEL-LINUX-LD -T
    /USR/LOCAL/SHARE/UMPS2/ELF32LTSMIP.H.UMPSCORE.X
    /USR/LOCAL/LIB/UMPS2/CRTSO.O P2TEST.O
    phase1 & phase2 .o files
    /USR/LOCAL/LIB/UMPS2/LIBUMPS.O -O KERNEL
```

The files ELF32LTSMIP.H.UMPSCORE.X, CRTSO.O, and LIBUMPS.O are part of the μMPS2 distribution. /USR/LOCAL/XXXX/UMPS2/ 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.[2]

The linker produces a file in the ELF object file format which needs to be converted prior to its use with μMPS2. This is done with the command:

```
UMPS2-ELF2UMPS -K KERNEL
```

[1] The recommended installation location for P2TEST.C along with a sample Makefile is /USR/LOCAL/SHARE/KAYA/

[2] As documented in Section 8.1-*pops*, if one is working on a big-endian machine one should modify the above commands appropriately; substitute MIPS- for MIPSEL- above.

which produces the file: KERNEL.CORE.UMPS

Finally, your code can be tested by launching μMPS2. Entering:

UMPS2

without any parameters loads the file KERNEL.CORE.UMPS by default. See Chapter 9-*pops*for details on using the μMPS2 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 μMPS2 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, μMPS2 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.

If you want to travel around the world and be invited to speak at a lot of different places, just write a Unix operating system.

Linus Torvalds

4

Phase 3 - Level 4: The VM-I/O Support Level

Level 4 (the VM-I/O support level) of Kaya builds on the nucleus level in four key ways to create an environment for the execution of user-processes (U-proc's):

- Support for virtual memory (VM). Each U-proc will run with **Status.VMc**=1 in its own virtual memory space using a unique ASID.
- Support for accessing the various I/O devices. In particular, U-proc's will be loaded from tape devices and one of the disk devices will be used as the backing store for the VM implementation. U-proc's will have read/write access to the other disk devices, write access to the printers and terminals and optionally read access to the terminals as well.
- To facilitate U-proc cooperation, this level will provide high-level process synchronization primitives; a P and V service that supports virtual addresses.
- Support for a process delay facility. U-proc's, in addition to being able to access the TOD clock, will have the capability to "sleep" for a specified period of time. i.e. A U-proc can request that it be removed from the Ready Queue for the specified period of time.

Another perspective on the VM-I/O support level is that by building on the exception handling facility of the nucleus, this level provides the U-proc-level exception handlers that the nucleus exception handlers "passes" exception handling "up" to; assuming that the appropriate SYS5 for the exception type was performed. There will be one exception handler for each type of exception:

- Program Trap (PgmTrap) exceptions: This exception handler will terminate the process in a controlled manner.

- SYSCALL/Breakpoint (SYS/Bp) exceptions: This exception handler will implement the new SYS services the VM-I/O support level exports/implements.

- TLB Management (TLB) exceptions: This exception handler will implement Kaya's virtual memory support; i.e. the system Pager.

These exception handlers will run in kernel-mode with **Status.VMc**=1, while the U-proc's will run in user-mode with **Status.VMc**=1. Hence each U-proc leads a schizophrenic life, mostly executing in user-mode, but sometimes, after the handling of an exception is "passed" back up to it, executing in kernel-mode. While the nucleus exception and interrupt handlers are system-wide resources that all processes share (in serial fashion), the VM-I/O support level exception handlers are more like VM-I/O support level provided libraries that becomes part of each U-proc.

The overall purpose of the VM-I/O support level is to provide a simple time-sharing system which will support up to eight U-proc's. `UPROCMAX` should be defined to the specific degree of multiprogramming to be supported/implemented: [1...8]. Associated with each U-proc will be a terminal, a printer, and a tape device which will hold the U-proc's executable image. U-proc's will each execute with **Status.VMc**=1 and in their own unique virtual address space. U-proc's, which run in user-mode with interrupts enabled, are considered untrustworthy. Nonetheless U-proc's need a way to request system services in a safe way that does not compromise system security.

This suggests implementing new SYS operations. These are outlined in Section 4.1. However, several problems need to be addressed when providing this support:

- I/O — a process that can initiate I/O operations could specify any memory location for data transfer and can thus overwrite any location it wishes.
  Solution: make sure the page frame containing the device registers is not

accessible to U-proc's. Since the device registers reside in ksegOS and U-proc's run in user-mode with **Status.VMc**=1, access to the device registers is prevented by the hardware.

- Nucleus implemented SYS's — by specifying random locations as a semaphore, a process could alter any location it liked.
  Solution: run the U-proc's in user mode, since they are then prohibited from issuing nucleus implemented SYS's. This is why the nucleus specification contained this restriction.

- Arbitrary memory accesses using loads, stores, etc.
  Solution: use μMPS2's virtual memory support to give each U-proc a private address space mapped to kUseg2 using a unique ASID. This will give each U-proc an address space disjoint from both the VM-I/O support level's and the other U-proc's address spaces. Actually, Kaya provides for some pages to be shared by all U-proc's; the first n pages of kUseg3. A malicious U-proc could therefore interfere with other U-proc's that were using kUseg3 for synchronization purposes. This however is not a security hole, merely a nuisance.

It needs to be stressed that U-proc's must be assumed to be untrustworthy. Thus, it will be necessary for you to construct the VM-I/O support level so that it protects itself from U-proc's. For one thing, this will make this level easier to debug; U-proc's will be stopped before they (completely) destroy the integrity of the system.

The VM-I/O support level's functions, which are considered to be trustworthy, will run in kernel-mode with interrupts unmasked. The code and data for these functions will reside in the address space of the kernel. Probably the trickiest aspect of the VM-I/O support level is that the functions of this level must be able to access the private data for each U-proc.

As described in Section 3.2 the nucleus, after initialization, starts the system by creating a single process: user-mode off, interrupts enabled, **Status.VMc**=0, **Status.TE**=1, **$SP** set to the penultimate RAM frame, and **PC**=`test`. The VM-I/O support level initialization function should therefore be called `test`.

4.1 SYS/Bp Exception Handling

As described in Section 3.3, the nucleus directly handles all SYS1-SYS8 SYSCALL exceptions and Breakpoint exceptions [1. . .4] (**LDST**, **FORK**, **PANIC**, and **HALT**).

For all other SYSCALL and Breakpoint exceptions the nucleus either treats the exception as a SYS2 or if the offending process has issued a SYS5 for SYS/Bp exceptions "passes up" the handling of the exception. Assuming that the handling of the exception is to be passed up, that the U-proc's SYS/Bp New Area was correctly initialized, and a SYS5 for SYS/Bp exceptions was performed during U-proc initialization, execution continues with the VM-I/O support level's SYS/Bp exception handler. The processor state at the time of the exception will be in the U-proc's SYS/Bp Old Area.

A SYSCALL exception is distinguished from a Breakpoint exception by the contents of **Cause.ExcCode** in the U-proc's SYS/Bp Old Area. SYSCALL exceptions are recognized via an exception code of *Sys* (8) while Breakpoint exceptions are recognized via an exception code of *Bp* (9).

By convention the executing process places appropriate values in user registers **a0**– **a3** immediately prior to executing a **SYSCALL** or **BREAK** instruction. The VM-I/O support levelSYS/Bp exception handler will then perform some service on behalf of the U-proc executing the **SYSCALL** or **BREAK** instruction depending on the value found in **a0**.

In particular, if a U-proc executes a **SYSCALL** instruction and **a0** contained a value in the range [9..18] then the VM-I/O support level should perform one of the services described below.

4.1.1 Read_From_Terminal (SYS9)

`int SYS9 (READ_FROM_TERMINAL, char *addr)` When requested, this service causes the requesting U-proc to be suspended until a line of input (string of characters) has been transmitted from the terminal device associated with the U-proc.

The SYS9 service is requested by the calling U-proc by placing the value 9 in **a0**, the virtual address of a string buffer where the data read should be placed in **a1**, and then executing a **SYSCALL** instruction. Once the process resumes the number of characters actually transmitted is returned in **v0** if the read was successful. If the operation ends with a status other than "Character Received" (5), the negative of the device's status value is returned in **v0**.

Attempting to read from a terminal device to an address in ksegOS is an error and should result in the U-proc being terminated (SYS18).

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

```
int SYSCALL (READTERMINAL, char *virtAddr
```

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

4.1.2 Write_To_Terminal (SYS10)

When requested, this service causes the requesting U-proc to be suspended until a line of output (string of characters) has been transmitted to the terminal device associated with the U-proc.

The SYS10 service is requested by the calling U-proc by placing the value 10 in **a0**, the virtual address of the first character of the string to be transmitted in **a1**, the length of this string in **a2**, and then executing a **SYSCALL** instruction. Once the process resumes the number of characters actually transmitted is returned in **v0** if the write was successful. If the operation ends with a status other than "Character Transmitted" (5), the negative of the device's status value is returned in **v0**.

It is an error to write to a terminal device from an address in ksegOS, request a SYS10 with a length less than 0, or a length greater than 128. Any of these errors should result in the U-proc being terminated (SYS18).

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

```
int SYSCALL (WRITETERMINAL, char *virtAddr, int len)
```

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

4.1.3 V_Virtual_Semaphore (SYS11)

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 SYS11 service is requested by the calling U-proc by placing the value 11 in **a0**, the *virtual* address of the semaphore to be V'ed in **a1**, and then executing a **SYSCALL** instruction.

Attempting to perform a V operation on an address in ksegOS or kUseg2 is an error and should result in the U-proc being terminated (SYS18).

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

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

Where the mnemonic constant `VSEMVIRT` has the value of 11.

4.1.4 P_Virtual_Semaphore (SYS12)

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

The P or SYS12 service is requested by the calling U-proc by placing the value 12 in **a0**, the *virtual* address of the semaphore to be P'ed in **a1**, and then executing

a **SYSCALL** instruction.

Attempting to perform a P operation on an address in ksegOS or kUseg2 is an error and should result in the U-proc being terminated (SYS18).

The following C code can be used to request a SYS12:

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

Where the mnemonic constant `PSEMVIRT` has the value of 12.

4.1.5 Delay (SYS13)

This service causes the executing U-proc to be delayed for n seconds. The requesting U-proc is to be delayed at least n seconds and not substantially longer. Since the nucleus controls low-level scheduling decisions, all the VM-I/O support level can ensure is that the requesting U-proc not be "schedulable" until n seconds has elapsed and that it becomes schedulable shortly thereafter.

The Delay or SYS13 service is requested by the calling U-proc by placing the value 13 in **a0**, the number of seconds to be delayed in **a1**, and then executing a **SYSCALL** instruction.

Attempting to request a Delay for less than 0 seconds is an error and should result in the U-proc begin terminated (SYS18).

The following C code can be used to request a SYS13:

```
void SYSCALL (DELAY, int secCnt)
```

Where the mnemonic constant `DELAY` has the value of 13.

4.1.6 Disk_Put (SYS14) and Disk_Get (SYS15)

These services provide *synchronous* I/O on the μMPS2 disk devices. When requested, this service causes the requesting U-proc to be suspended until the disk write (or read) operation has concluded.

The SYS14 (SYS15) service is requested by the calling U-proc by placing the value 14 (15) in **a0**, the virtual address of the 4KB block to be written to the disk (to contain the data from the disk) in **a1**, the disk number ([1. . .7]) in **a2**, the disk sector to be written onto (read from) in **a3**, and then executing a **SYSCALL** instruction. Once the process resumes **v0** is to contain the completion status of the disk operation. If the operation ends with a status other than "Device Ready" (1), the negative of the completion status is returned in **v0**.

From one perspective disk devices are three dimensional devices: cylinders (or tracks), surfaces (or heads) and sectors. From another perspective they are only one-dimensional: sectors. A disk device with x cylinders, y surfaces, and z

sectors/track can be thought of being a (one dimensional) device with $sectCnt = x * y * z$ sectors numbered $[0 \ldots (sectCnt - 1)]$. The SYS14/SYS15 services, since they only take a disk sector parameter (instead of a disk sector, surface#, and track#) assumes this one dimensional perspective for disk devices.

Attempting to write to (read from) a disk device from (into) an address in ksegOS is an error and should result in the U-proc being terminated (SYS18). Similarly, attempting to perform a disk operation upon DISK0, which is reserved for use by the VM implementation as the backing store device is an error and should result in the U-proc being terminated (SYS18).

The following C code can be used to request a SYS14:

```
   int SYSCALL (DISK_PUT, int *blockAddr, int diskNo,
int sectNo)
```

Where the mnemonic constant `DISK_PUT` has the value of 14. A SYS15 is requested by substituting `DISK_PUT` with `DISK_GET`, where the mnemonic constant `DISK_GET` has the value of 15.

4.1.7 Write_To_Printer (SYS16)

When requested, this service causes the requesting U-proc to be suspended until a line of output (string of characters) has been transmitted to the printer device associated with the U-proc.

Once the process resumes the number of characters actually transmitted is returned in **v0**.

The SYS16 service is requested by the calling U-proc by placing the value 16 in **a0**, the virtual address of the first character of the string to be transmitted in **a1**, the length of this string in **a2**, and then executing a **SYSCALL** instruction. Once the process resumes the number of characters actually transmitted is returned in **v0** if the write was successful. If the operation ends with a status other than "Device Ready" (1), the negative of the device's status value is returned in **v0**.

It is an error to write to a printer device from an address in ksegOS, request a SYS16 with a length less than 0, or a length greater than 128. Any of these errors should result in the U-proc being terminated (SYS18).

The following C code can be used to request a SYS16:

```
   int SYSCALL (WRITEPRINTER, char *virtAddr, int len)
```

Where the mnemonic constant `WRITEPRINTER` has the value of 16.

4.1.8 Get_TOD (SYS17)

When this service is requested, it causes the number of microseconds since the system was last booted/reset to be placed/returned in the U-proc's **v0**.

The SYS17 service is requested by the calling U-proc by placing the value 17 in **a0** and then executing a **SYSCALL** instruction.

The following C code can be used to request a SYS17:

```
unsigned int SYSCALL (GETTOD)
```

Where the mnemonic constant `GETTOD` has the value of 17.

4.1.9 Terminate (SYS18)

This services causes the executing U-proc to cease to exist.

When all U-proc's have terminated, Kaya should "shut down." Thus, somehow the "system" processes created in the VM-I/O support level (e.g. the delay daemon process – see Section 4.3) must be terminated after all the U-proc's have terminated. Since there should then be no dispatchable or blocked processes, the nucleus scheduler will invoke the **HALT** ROM service/instruction. (See Section 3.1.)

The SYS18 service is requested by the calling process by placing the value 18 in **a0** and then executing a **SYSCALL** instruction.

The following C code can be used to request a SYS18:

```
void SYSCALL (TERMINATE)
```

Where the mnemonic constant `TERMINATE` has the value of 18.

4.2 PgmTrap Exception Handling

As described in Section 3.4 the nucleus either treats a PgmTrap exception as a SYS2 or if the offending process has issued a SYS5 for PgmTrap exceptions "passes up" the handling of the exception. Assuming that the handling of the exception is to be passed up, that the U-proc's PgmTrap New Area was correctly initialized, and a SYS5 for PgmTrap exceptions was performed during U-proc initialization, execution continues with the VM-I/O support level's PgmTrap exception handler. The processor state at the time of the exception will be in the U-proc's PgmTrap Old Area.

The VM-I/O support level's PgmTrap exception handler is to terminate the process in an orderly fashion; perform the same operations as a SYS18 request.

4.3 Delay Facility

The SYS13 Delay facility allows a requesting U-proc to be "put to sleep" for a specified number of seconds. A process that is neither the current process nor sitting on the Ready Queue can be considered to be "sleeping." There are two issues that need addressing for the implementation of this facility: where to place the U-proc while it is sleeping, and how to keep track of which U-proc's are sleeping so they can be awoken (i.e. placed on the Ready Queue) at the appropriate time.

4.3.1 Where to Store Sleeping U-proc's

As mentioned above, access to the nucleus is limited solely to requesting SYS1-SYS8 services. Therefore the only way to put a U-proc to sleep (i.e. keep it off of the Ready Queue) is to block the U-proc on a semaphore. The VM-I/O support level should therefore contain an array of size `UPROCMAX` of semaphores; one for each U-proc. These semaphores are defined as the *U-proc private semaphores*. As part of U-proc initialization each U-proc's private semaphore should be initialized to zero so that a P operation on this semaphore will cause the U-proc to block.

Hence the VM-I/O support level should interpret a SYS13 as a request to perform a P operation on the U-proc's private semaphore.

4.3.2 Keeping Track of Sleeping U-proc's

The VM-I/O support level needs to maintain a list of sleeping U-proc's. The following implementation is suggested: Maintain a sorted NULL-terminated single linearly linked list (using the `d_next` field) of delay-node descriptors whose head is pointed to by the variable `delayd_h`. The list `delayd_h` points to will represent the list of pending "wake up calls;" the *Active Delay* List (ADL). Keep the ADL sorted in ascending order using the `d_wakeTime` field as the sort key.

Maintain a second list of delay-node descriptors, the *delaydFree* list, to hold the unused delay-node descriptors. This list, whose head is pointed to by the variable `delaydFree_h`, is kept, like the pcbFree and the semdFree lists, as a NULL-terminated single linearly linked list (using the `d_next` field).

The delay-node descriptors themselves should be declared, like the ProcBlk's and semaphore descriptors, as a static array of size `UPROCMAX` of type `delayd_t`. In addition to the `d_next` pointer and `d_wakeTime` integer fields, a delay-node

descriptor should also contain an ASID integer field as well, to denote the sleeping U-proc's identity.

When a U-proc requests some "quiet time," in addition to performing a P operation on the U-proc's private semaphore, a delay-node needs to be allocated from the delaydFree list, populated with appropriate values, and inserted into the ADL.

Periodically, the VM-I/O support level needs to examine the ADL to determine if a U-proc's wake time has passed. To accomplish this the VM-I/O support level will implement a special VM-I/O support level process (i.e. a daemon); the delay daemon process. The delay daemon process will repeat forever

1. Request a Wait_For_Clock (SYS7) nucleus service.

2. Upon resumption of execution, examine the ADL, removing all delay-nodes whose wake time has passed. For each delay-node whose wake time has passed, perform a V operation on the indicated U-proc's private semaphore and return the delay-node to the delaydFree list.

Therefore, the delay process will wake every 100 milliseconds (i.e. a pseudo-clock tick event), examine the ADL, waking up U-proc's if their delay has expired, and then return to sleep (SYS7). The delay daemon process will run in kernel-mode using a unique ASID with **Status.VMc**=1 and all interrupts enabled.

4.4 Virtual P and V Service

The SYS11/SYS12 P & V facility for virtual addresses allows a requesting U-proc to request a P or V operation on a semaphore with a virtual address. Since U-proc's run in user-mode and are restricted to only using virtual addresses, the nucleus SYS3/SYS4 service will not be of use to U-proc's wishing to coordinate their cooperation through use of semaphores. As with the Delay Facility, there are two issues that need addressing for the implementation of this facility: where to place a U-proc blocked on a virtual-addressed semaphore, and how to keep track of which U-proc's are blocked on a given semaphore so that they can be awoken (i.e. placed on the Ready Queue) at the appropriate time; when a V operation is requested for the specified semaphore.

4.4.1 Where to Store Blocked U-proc's

When a U-proc performs a P (SYS12) operation on a virtual-addressed semaphore and the value of the semaphore becomes ≤ 0 (i.e. the U-proc is to be blocked), the VM-I/O support level should interpret this as a request to perform a P operation on the requesting U-proc's private semaphore.

4.4.2 Keeping Track of Blocked U-proc's

The VM-I/O support level needs to maintain a list of U-proc's blocked because of a SYS12 operation. The following implementation is suggested: maintain a double circularly linked list (using the `vs_next` and `vs_prev` fields) of virtSem-node descriptors whose head is pointed to by the variable `virtSemd_h`. The list `virtSemd_h` points to will represent the list of U-proc's blocked because of a SYS12 operation; the *Active Virtual Semaphore* List (AVSL).

Maintain a second list of virtSem-node descriptors, the *virtSemdFree* list, to hold the unused virtSem-node descriptors. This list, whose head is pointed to by the variable `virtSemdFree_h`, is kept, like the delaydFree, pcbFree, and semdFree lists, as a NULL-terminated single linearly linked list (using the `vs_next` field).

The virtSem-node descriptors themselves should be declared, like the delay-node, ProcBlk, and semaphore descriptors, as a static array of size MAXPROC of type `virtSemd_t`. In addition to the `vs_next` and `vs_prev` pointer fields, a virtSemd-node descriptor should also contain a `vs_semaphore` integer field, and an ASID integer field, to denote the blocked U-proc's identity.

When a U-proc is to be blocked as a result of a SYS12 request, in addition to performing a P operation on the U-proc's private semaphore, a virtSem-node needs to be allocated from the virtSemdFree list, populated with appropriate values, and inserted into the AVSL.

When a U-proc is to be unblocked as a result of a SYS11 request, a search is made of the AVSL for a virtSem-node with a matching `vs_semaphore` field. This node is removed from the AVSL and returned to the virtSemdFree list. Finally a V operation is performed on the unblocked U-proc's private semaphore.

Remember that insertion into and the search for removal from the AVSL should be performed so that when there is more than one U-proc blocked because of a SYS12 on the same semaphore the order of unblocking/removal is first-in first-out.

4.5 Implementing Virtual Memory

As described in Section 3.5 the nucleus either treats a TLB exception as a SYS2 or if the offending process has issued a SYS5 for TLB exceptions "passes up" the handling of the exception. Assuming that the handling of the exception is to be passed up, that the U-proc's TLB New Area was correctly initialized, and a SYS5 for TLB exceptions was performed during U-proc initialization, execution continues with the VM-I/O support levelś TLB exception handler; the *Pager*. The processor state at the time of the exception will be in the U-proc's TLB Old Area.

There are a number of situations that trigger a TLB exception; see Chapter 4-*pops*. Which of these exception types are to be handled by the Pager and which are to trigger a SYS18 depend on the sophistication of the Pager to be implemented. This section describes a very basic Pager. Throughout the following sub-Sections are suggestions for improving/expanding the Pager.

4.5.1 System Segment Tables

As described in Section 4.3.1-*pops*, each ASID's segment table is located in the ROM Reserved Frame. For a given ASID the segment table holds three entries; the addresses of the ksegOS, kUseg2, and kUseg3 PTE's. These must be initialized during U-proc initialization.

All U-proc ASID's will share the same ksegOS PTE; there is one ksegOS PTE and all ASID ksegOS PTE segment table pointers will point to it. The single ksegOS PTE is a VM-I/O support level data structure. This segment table entry is only used when the U-proc generates a ksegOS reference when in kernel-mode, since any reference to ksegOS while in user-mode generates an Address Error exception.

All U-proc ASID's will share the same kUseg3 PTE; there is one kUseg3 PTE and all ASID kUseg3 segment table pointers will point to it. The single kUseg3 PTE is a VM-I/O support level data structure.

Each U-proc ASID will have its own kUseg2 PTE. The array of kUseg2 PTE's is also a VM-I/O support level data structure.

Finally, the segment table entries for the delay daemon process will have the same ksegOS entry above and NULL for both the kUseg2 and kUseg3 entries.

4.5.2 U-proc Page Tables

As described above the VM-I/O support level needs to implement `UPROCMAX+2` PTE's; one for each U-proc and one each for the ksegOS and kUseg3 segments. Exclusive of the ksegOS PTE, each PTE should contain `MAXPAGES` entries (where `MAXPAGES` = 32).

For the kUseg3 PTE, the `MAXPAGES` entries should describe the first `MAXPAGES` pages of the segment (0xC000.0000 – 0xC001.F000). Each page's PTE entry in **EntryHi** should indicate the **VPN** and **SEGNO** of the page, and the ASID should be set to zero. Each page's PTE entry in **EntryLo** should indicate that the entry is global, but invalid (i.e. not present).

For the individual kUseg2 PTE's, the `MAXPAGES` entries should describe the first `MAXPAGES-1` pages of the segment (0x8000.0000 – 0x8001.E000) and the last page of the segment (0xBFFF.F000). Each page's PTE entry in **EntryHi** should indicate the **VPN** and **SEGNO** of the page, and the ASID should be set to the ASID of the U-proc. Each page's PTE entry in **EntryLo** should indicate that the entry is both not global and invalid.

As the above definitions illustrate, Kaya is designed to only support U-proc's whose combined **.text**, **.data**, and **.bss** areas never grow beyond `MAXPAGES-1` pages, whose stack needs never exceed one page, and whose utilization of kUseg3 is always within the segments first `MAXPAGES` pages.

The Layout of ksegOS and its PTE

The installed physical RAM begins at 0x2000.0000 and goes up to RAMTOP. The default bootstrap loaders load the OS beginning at 0x2000.1000; the first frame of RAM is reserved for the ROM Reserved Frame. As described in Section 8.3-*pops* the OS **.text** and **.data** areas are loaded adjacent to each other starting at 0x2000.1000. The stack frame for the nucleus is the last frame of RAM, while the stack frame for `test` is the penultimate frame of RAM. Additionally:

- Each U-proc needs two RAM frames, (`UPROCMAX` * 2), for stack space; one each for its VM-I/O support level SYS/Bp and TLB exception handlers. The stack page for when a U-proc is running in user-mode is the last page of kUseg2, a virtual page which will get placed somewhere in the page pool.
- The delay daemon process needs one RAM frame for its stack space.
- Each DMA-supporting I/O device needs a RAM frame for its I/O buffer. The reason for this is covered in Section 4.8.1.

- (UPROCMAX * 2) frames need to be reserved for VM paging. In spite of the additional RAM, the page pool is limited to (UPROCMAX * 2) frames to force paging events.

All of the above frames, except the (UPROCMAX * 2) frames comprising the page pool and the two stack frames located below RAMTOP need to be located below 0x8000.0000. Figure 4.1 illustrates one suggested organization.

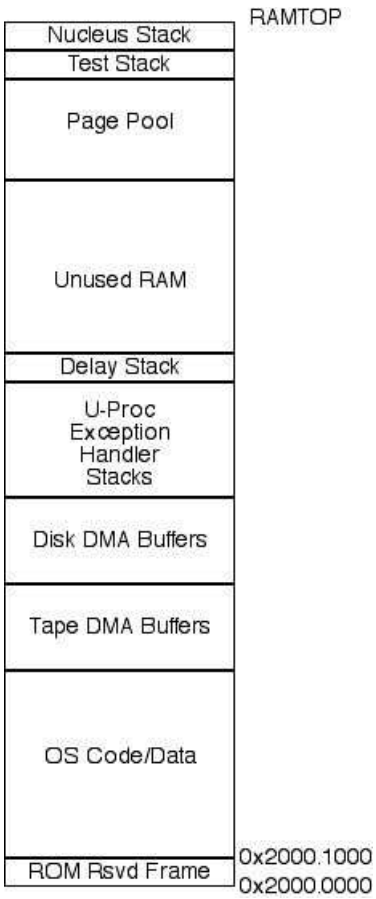

Figure 4.1: RAM Frame Layout Organization

The single ksegOS PTE needs to be set up so that the VM-I/O support level exception handlers can run with **Status.VMc**=1, but that no page fault ever occurs for an address in ksegOS, and all addresses generated in ksegOS get translated into the same physical address that would be generated if **Status.VMc**=0. The VM-I/O support level exception handlers need to run with **Status.VMc**=1 so that the

U-proc's kUseg2 (and kUseg3) address space(es) are addressable. Furthermore, the VM-I/O support level exception handlers need to run as if **Status.VMc**=0 since these handlers interact with the nucleus and device registers which only understand physical addresses.

To accomplish this one should allocate a ksegOS PTE of approximately 50 entries. These entries will describe the 50 physical frames beginning at 0x2000.0000. Each page's PTE entry in **EntryHi** should indicate the **VPN** and **SEGNO** of the page, and the ASID should be set to zero. Each page's PTE entry in **EntryLo** should indicate that the entry is global, writable (i.e. dirty), valid, and located at its corresponding **PFN**. (e.g. The page at **SEGNO**=0, **VPN**=0x20000 would have a **PFN**=0x20000.)

The first n entries describe the **.text** and **.data** areas of the OS in addition to the ROM Reserved Frame. The next 16 entries describe the 8 DMA disk buffers and the 8 DMA tape buffers. The final (`UPROCMAX` * 2) + 1 entries describe the stack frames for the VM-I/O support level exception handlers (2 per U-proc) and one for the delay daemon process.

Interestingly the ksegOS PTE entries describing frames that contain nucleus **.text** and **.data** areas can be either omitted from the PTE or at least marked as invalid without affecting the correct performance of the OS. Since the nucleus runs with **Status.VMc**=0 it makes no use of the ksegOS PTE. The VM-I/O support level exception handlers do not directly call nucleus functions or access nucleus variables/data structures; all interaction is via the SYS1-SYS8 nucleus services. Individuals wishing to implement a more sophisticated Pager should attempt this experiment.

4.5.3 The Swap Pool

(`UPROCMAX` * 2) physical frames are reserved for paging purposes. While there is probably sufficient available RAM to use more than (`UPROCMAX` * 2) frames for paging, limiting the page pool to (`UPROCMAX` * 2) frames will insure that page faults will occur. The (`UPROCMAX` * 2) frames that are to be used for the page pool can be located anywhere in RAM, excepting of course the first 50 initial RAM frames or the final two RAM frames. It is recommended that the (`UPROCMAX` * 2) contiguous frames preceeding `test`'s stack frame be used for the page pool.

The VM-I/O support level needs to implement a data structure describing the page pool. For each frame in the pool, one needs to record whether the frame is in use or not, and if so, by which U-proc (i.e. ASID) and which virtual page is occupying the frame (**SEGNO**, and **VPN**).

4.5.4 Handling a TLB Invalid Exception

As described in Section 4.3.4-*pops* all TLB-Refill event's are handled by the ROM-TLB-Refill handler. As for TLB exception types, there are four to consider:

- TLB-Modification(*Mod*): This exception should never occur under normal processing since all valid PTE entries should always be marked as dirty.
- Bad-PgTbl(*BdPT*): This exception should never occur under normal processing since it is hoped that all PTE's will be correctly constructed.
- PTE-MISS(*PTMs*): This exception should never occur under normal processing since if correctly constructed, each PTE contains all the necessary entries. Some of these entries will be for entries that are invalid, but the entry should nonetheless be present in the PTE.
- TLB-Invalid(*TLBL* & *TLBS*): This is the only exception that the Pager should be designed to handle. This exception indicates a "page missing" page fault. All other TLB exception types should trigger a SYS18 response.

Handling a **TLB-Invalid** exception involves:

1. Determining the **SEGNO** and **VPN** of the offending address. This will be found in the U-proc's TLB Old Area.
2. Gain mutual exclusive access to the paging data structures. More on this can be found in Section 4.8.2.
3. Determine if the missing page is still missing. A page fault for a page in the shared segment (kUseg3) may have been brought into RAM by another U-proc while the current U-proc was waiting for mutual exclusion to be granted. If the missing page is no longer missing, release mutual exclusion and return control to the U-proc's execution stream.
4. Select a frame to be used for this page fault. For our simple Pager it is sufficient to use the "Oldest Page First" frame selection algorithm. Some refer to this algorithm as the "Round Robin" frame selection algorithm; first select frame 0, then 1, 2, . . ., 8, 9, 0, 1, etc.
5. If the frame is occupied mark the appropriate PTE entry to indicate that the entry is invalid. A copy of this entry may be sitting in the TLB. It

is necessary to mark this entry as invalid as well. The simplest way to accomplish this is to issue a **TLBCLR** instruction. (See Section 6.3.1-*pops* on how to do this in C.) Those wishing to implement a more sophisticated Pager may optionally perform a TLB-Probe to find the matching entry in the TLB, and if present to alter the entry in the TLB.

Regardless, altering both the designated PTE entry and the TLB must be done atomically. (Remember that the VM-I/O support level exception handlers run with interrupts enabled.) An error can occur if the U-proc is interrupted after having updated the PTE but before updating the TLB. (Note: Given the load/store nature of RISC architectures like μMPS2, it is also insufficient to simply update the TLB first.) To alter both atomically one should disable interrupts (i.e. mask them by setting **Status.IEp**=0) before the two updates and then re-enable them after the two updates.

6. If the frame was occupied, assume it is dirty and write it out to the backing store device (DISK0). The backing store device needs to reserve `MAXPAGES` sectors (or blocks) for each U-proc plus an additional `MAXPAGES` sectors for pages from kUseg3. How the backing store device's sectors are divided up among the U-proc's and kUseg3 is left up to the OS author. There is no need to use a DMA buffer for this disk write since the physical RAM address is known and will not change mid-write; the executing U-proc holds mutual exclusion.

7. Read in the missing page from the backing store device. Where it is found on the device is left up to the OS author; see above point. As with the disk write, there is no need to use a DMA buffer for this disk read since the physical RAM address is known and will not change mid-read; the executing U-proc still holds mutual exclusion.

8. Update the swap pool data structure to reflect the new occupant of the given frame in the page pool.

9. Update the appropriate PTE to reflect that the page is now sitting in RAM. (i.e. Mark the **EntryLo** field in the page's PTE to indicate that the entry is writable (i.e. dirty), valid, and located at the selected **PFN**.) The TLB also needs to be updated as well. Since a TLB-Invalid exception can only occur when there was a matching entry in the TLB at the time of the exception, not updating the TLB might lead to an infinite loop of TLB-Invalid exceptions.

Furthermore, as with flushing the previous frame's contents to the backing store device, updating the PTE and the TLB needs to be done atomically.

The simplest way to update the TLB (i.e. flush the TLB of its invalid entry) is to issue a **TLBCLR** instruction. Those wishing to implement a more sophisticated Pager may optionally perform a TLB-Probe to find the matching entry in the TLB, and if present to alter the entry in the TLB.

10. Release mutual exclusive access to the paging data structures and return control to the U-proc's execution stream; μMPS2's re-attempt to translate a virtual address into a physical one.

4.5.5 A More Sophisticated Pager

As described above, there are a number of improvements those wishing to implement a more sophisticated Pager can take; mark frames containing nucleus **.text** and **.data** areas as not present and/or update a TLB entry directly instead of invalidating the complete TLB. Orthogonally, one may mark frames containing U-proc **.text** pages as read-only and U-proc **.data** pages as writable.

Another improvement would be to relax the assumption that all pages are dirty and need to be written to the backing store device. Unfortunately, μMPS2 does not automatically update a "dirty" bit whenever a page is written to. One can nonetheless simulate this using an auxiliary data structure. Whenever a page is brought into RAM do not mark its PTE **EntryLo** entry as dirty. Therefore, whenever a U-proc attempts to write on such a page a TLB-Modification TLB exception will occur. Now, instead of performing a SYS18 on the offending U-proc, the fact that the page is now dirty must be recorded in the auxiliary data structure. Furthermore, the dirty bit in both the relevant PTE and TLB entries need to be turned on as well (atomically of course). Now when a selected frame is to be vacated for an incoming page, its current contents only need to be written out to the backing store device if indeed the page was dirty.

Finally, the "Oldest Page First" frame selection algorithm can be replaced with something a bit more sophisticated.

4.6 VM-I/O Support Level Initialization

After the nucleus concludes its initialization, control passes to `test`. This process/routine has a number of important tasks to complete:

1. Initialize the single ksegOS PTE.
2. Initialize the single kUseg3 PTE.
3. Initialize all VM-I/O support level semaphores.
4. Initialize the ADL module.
5. Initialize and launch the delay daemon process; this includes initializing the appropriate entries in its segment table.
6. Initialize the AVSL module.
7. Initialize the swap-pool data structure(s).
8. Initialize and launch each U-proc. See Section 4.7.
9. Go to sleep until all U-proc's have terminated. One way to accomplish this is to block `test`) on a semaphore (i.e. the `masterSem`) until all the U-proc's have terminated. Since U-proc termination is performed in a controlled manner; SYS18, it is a simple matter to know when the last U-proc has terminated. When this happens, the U-proc termination routine can simply V the `masterSem`, unblocking `test`.
10. Invoke the SYS2 Terminate service to halt the OS.

4.7 U-proc Initialization

Launching a U-proc is a complicated two-step process.

4.7.1 U-proc Initialization: Step 1 - variable initialization

The first step in launching a U-proc is to initialize various data structures and perform a SYS1 operation:

1. Assign the U-proc a unique ASID.
2. Initialize the U-proc's kUseg2 PTE.
3. Initialize the U-proc's private semaphore.

4. Initialize the U-proc's segment table.

5. Initialize the U-proc's three (PgmTrap, TLB, and SYS/Bp) New (processor state) Areas. The six processor state areas to be used by each U-proc when it makes its three SYS5 requests are yet another VM-I/O support level implemented data structure. Each New Area should be initialized to be a state with interrupts enabled, user-mode off, **Status.TE**=1, and **Status.VMc**=1. The **$SP** should be set to the appropriate stack page reserved for this U-proc's SYS/Bp or TLB exception handler respectively. Finally set the **PC** (and **t9**) to the address of the respective VM-I/O support level exception handler and **EntryHi.ASID** to the U-proc's assigned ASID.

6. Since it is imperative that all SYS5 requests be made while in kernel-mode and before virtual memory is enabled one needs to initialize a processor state appropriate for this goal. Initialize a processor state such that interrupts are enabled, user-mode is off, **Status.TE**=1, and **Status.VMc**=0. The **$SP** should be set to one of the stack pages reserved for this U-proc's SYS/Bp or TLB exception handler. Finally set the **PC** (and **t9**) to the address of the U-proc step 2 initialization function and **EntryHi.ASID** to the U-proc's assigned ASID.

7. Perform a SYS1 operation using the state in the above step.

4.7.2 U-proc Initialization: Step 2 - SYS5 Requests & Reading From the Tape

The second step in launching a U-proc is to issue the three SYS5 requests, read in the U-proc's **.text** and **.data** from the tape, and prepare for actual U-proc launch.

1. Determine the running U-proc's ASID; Perform a `getENTRYHI` and extract out the ASID value.

2. Perform the three SYS5 operations.

3. Read in the U-proc's **.text** and **.data** areas into the U-proc's area on the backing store device from the tape device associated with this U-proc. The contents of the tape are to be read contiguously. The first block is page 0 for the U-proc's kUseg2, the second block is page 1, and so on. As described in Section 5.4-*pops* the end of a file on a tape is denoted by an **EOF** marker.

4. Prepare a processor state appropriate for the execution of a U-proc. To do this initialize a processor state such that interrupts are enabled, user-mode is on, **Status.TE**=1, and **Status.VMc**=1. The **$SP** should be set to the last page of kUseg2. Finally set the **PC** (and **t9**) to 0x8000.00B0 (i.e. The contents of the second word in kUseg2and **EntryHi.ASID** to the U-proc's assigned ASID. See Section 8.3-*pops* for an explanation as to why the **PC** (and **t9**) are set to this value instead of just the beginning of kUseg2.)

5. Perform a **LDST** operation using the state prepared in the above step to finally begin executing the code associated with this U-proc.

   Note: Immediately following this **LDST** the U-proc will experience two page faults. The first one will be for the stack page (the last page in kUseg2) and the second will be for the first page of code (the first page of kUseg2). The U-proc's PTE was initialized to indicate that neither was "present." The backing store device contains the **.text** and **.data** contents – initialized when the U-proc's tape contents were read in. (i.e. Initial contents of the first n pages of kUseg2.) The page on the backing store representing the stack page contains uninitialized junk - which is perfectly fine for a stack page for a newly created process.

   A nice but tricky optimization would be to avoid reading this stack page in from the backing store device for this page fault only. Similarly, when reading in page 0 from tape, in addition to writing it out to the backing store, also place it into a free frame.

4.8 Nuts and Bolts

4.8.1 Performing DMA I/O Operations

As described in Chapter 5-*pops* the disk and tape devices utilize DMA to read and write directly from/to RAM. The address for a DMA I/O operation, specified in the device's **DATA0** device register field, must be a physical address. I/O device controllers operate independently of the **CP0** co-processor and by extension the virtual memory address translation facility.

When a U-proc requests to read a block from a disk, the address of a contiguous physical 4KB area must be provided. Even assuming the U-proc supplied virtual address is currently in RAM and that the page containing this address can be locked into its current frame (yet another level of sophistication for the Pager),

the 4KB block will likely spill over into the next page in the virtual address space. There is no guarantee that the frame holding the succeeding page, if even present, sits immediately after the frame holding the page containing the beginning of the block.

Instead the VM-I/O support level will provide an individual 4KB buffer for each DMA supporting device. (See Section 4.5.2.) The DMA supporting devices will read/write directly from/to their assigned buffer. It is the task of the VM-I/O support level I/O routines to copy the data to/from these buffers from/to the specified virtual address space.

For example for a disk write request the VM-I/O support level I/O routine would, after validating the virtual address, copy the 4KB from the virtual address space into the designated device's assigned buffer. After this is done, the disk write operation can proceed.

Note that the VM-I/O support level's I/O routine (part of the SYS/Bp exception handler) will be running with **Status.VMc**=1 and its ASID set to the current U-proc. From one perspective the copy operation is from one virtual address (in kUseg2 or kUseg3) to another virtual address (in ksegOS). Any page faults that occur or the fact that the 4KB source block is not necessarily contiguous in RAM is automatically taken care of by virtual address translation. From another perspective, given the way the ksegOS PTE was constructed, the copy operation is a from a virtual address to a physical address.

Note: It is not necessary to use the VM-I/O support level DMA buffer for paging related disk I/O. Paging I/O always begins on a frame boundary and because of mutual exclusion held by the process within the Pager, the physical RAM page is effectively locked. Hence paging related disk I/O can be performed directly to/from the physical frames in the page pool. This is also true for U-procinitialization, the buffer for a tape device (filled via a tape read operation) can be used as the source for backing store disk write operation.

4.8.2 Managing Concurrency

Level 4 of Kaya represents a timeshare system where many U-proc's along with some system daemons run concurrently. These processes can simultaneously be executing code in the same VM-I/O support level routine. There are two questions that need to be addressed: how can two U-proc's be executing code in the same VM-I/O support level routine at the same time; and how to prevent a race condition between two or more processes wishing to access the same VM-I/O support level data structure (e.g. the ADL, the AVSL, or the swap-pool data structure)?

The first question isn't an issue since each VM-I/O support level handler is implemented re-entrantly; each process that executes the code of one of these handlers has its own stack frame and hence its own copy of all local variables.

Race conditions can be explicitly avoided through the use of controlling semaphores. For each shared VM-I/O support level data structure there should be a semaphore defined for it that is initialized to one. Before an attempt to access (both read and write) a shared data structure one must first request a SYS4 operation on the data structure's controlling semaphore. The description in Section 4.5.4 provides an example of this for the swap-pool/Pager structures.

There should be one semaphore for the delay facility, the virtual P & V facility, the swap-pool/Pager service and one for each device's device registers. Remember that terminals are actually two independent sub-devices. Hence each terminal has two device register controlling semaphores.

4.8.3 Two Stacks per U-proc At The VM-I/O support level Explained

In the nucleus each exception handler is independent of each other. This is why all four nucleus exception handlers can use the same stack frame. At the VM-I/O support level it is possible for the VM-I/O support level SYS/Bp exception handler to generate a TLB exception (i.e. page fault). Consider a SYS11 (V) request where the increment of the semaphore causes a page fault since the page containing the semaphore is not present in RAM.

To handle these nested exceptions one needs independent stacks for the SYS/Bp and TLB exception handlers for each U-proc. The VM-I/O support level SYS/Bp exception handler generating a TLB exception is the only case where nested exceptions can occur though. The VM-I/O support level TLB exception handler will not make a SYSCALL that gets passed up, and hopefully neither the VM-I/O support level TLB or SYS/Bp exception handler will generate a PgmTrap exception. A separate stack is not needed for the VM-I/O support level PgmTrap exception handler. Either the VM-I/O support level's TLB or SYS/Bp exception handler's stack can be re-used as the VM-I/O support level's PgmTrap exception handler's stack frame.

4.8.4 The VM-I/O support level Exception Identity Question

When control passes to a VM-I/O support level's exception handler, the handler needs to know its ASID. Though there are separate stack frames for each U-proc, making the handlers reentrant, the code (i.e. *text*) is nonetheless the same for each U-proc. If during U-proc initialization (see Section 4.7) each U-proc initializes its three New (processor state) Areas to contain the U-proc's ASID, each VM-I/O support level exception handler, on entry, can easily learn its ASID. (Perform a `getENTRYHI` and extract out the ASID value.)

4.8.5 Module Decomposition

One possible module decomposition is as follows:

- INITPROC.C This module implements `test()` and all the U-proc initialization routines. It exports the VM-I/O support level's global variables. (e.g. swap-pool data structure, mutual exclusion semaphores, etc.)
- ADL.C This module implements the Active Delay List module.
- AVSL.C This module implements the Active Virtual Semaphore List module.
- PAGER.C This module implements the VM-I/O support level TLB exception handler; the Pager.
- SYSSUPPORT.C This module implements the VM-I/O support level SYS/Bp and PgmTrap exception handlers.

4.8.6 Accessing the `libumps` Library

As described in Chapter 6-*pops*, accessing the **CP0** registers and the ROM-implemented services/instructions in C is via the libumps library. Simply include the line

```
#include ''/usr/local/include/umps2/umps/libumps.e''
```

The file LIBUMPS.E is part of the μMPS2 distribution.
/USR/LOCAL/INCLUDE/UMPS2/UMPS/ is the recommended installation location for this file. Make sure you know where it is installed in your local environment and alter this compiler directive appropriately.

4.8.7 Testing

There is a set of provided test U-proc programs that will "exercise" your code.[1]

- SWAPTEST.C – Forces the page pool to fill to generate page faults.
- TODTEST.C – Tests the delay facility.
- DISKTEST.C – Tests U-proc disk I/O.
- PRINTERTEST.C – Tests writing to the printer.
- FIBTEST.C – processor intensive job; calculate Fib(7) recursively.
- READTEST.C - Tests for correct terminal input.
- PVTESTA.C & PVTESTB.C – Tests the VM-I/O support level P & V facility. These two programs must be run together.
- PRINT.C – A utility module used by all of the above test programs to facilite terminal printing.

Additionally, it is easy to write one's own programs to run under Kaya. Being able to run your own programs under your own OS is half the fun of completing the project anyway.

You should individually compile all the source files from phase1, phase2, and phase3 in addition to the phase3 U-proc test programs using the command:

```
MIPSEL-LINUX-GCC -ANSI -PEDANTIC -WALL -C FILENAME.C
```

All of the OS object files should then be linked together using the command:

```
MIPSEL-LINUX-LD -T
    /USR/LOCAL/SHARE/UMPS2/ELF32LTSMIP.H.UMPSCORE.X
    /USR/LOCAL/LIB/UMPS2/CRTSO.O
    phase1, phase2 & phase3 .o files
    /USR/LOCAL/LIB/UMPS2/LIBUMPS.O -O KERNEL
```

The linker produces a file in the ELF object file format which needs to be converted prior to its use with μMPS2. This is done with the command:

```
UMPS2-ELF2UMPS -K KERNEL
```

which produces the file: KERNEL.CORE.UMPS

[1]The recommended installation location for these test files along with a sample Makefile is /USR/LOCAL/SHARE/KAYA/

Each test program should individually be linked. The following is an example for SWAPTEST.O

```
MIPSEL-LINUX-LD -T
    /USR/LOCAL/SHARE/UMPS2/ELF32LTSMIP.H.UMPSAOUT.X
    /USR/LOCAL/LIB/UMPS2/CRTI.O PRINT.O SWAPTEST.O
    /USR/LOCAL/LIB/UMPS2/LIBUMPS.O -O SWAPTEST
```

The linker produces a file in the ELF object file format which needs to be converted prior to its use with μMPS2. This is done with the command:

```
UMPS2-ELF2UMPS -A SWAPTEST
```

which produces the file: SWAPTEST.AOUT.UMPS

Finally, the linked file can be loaded onto a tape cartridge with the commmand:

```
UMPS2-MKDEV -T SWAPTAPE.UMPS SWAPTEST.AOUT.UMPS
```

which produces the file: SWAPTAPE.UMPS

The files ELF32LTSMIP.H.UMPSCORE.X, ELF32LTSMIP.H.UMPSAOUT.X, CRTSO.O, CRTI.O, and LIBUMPS.O are part of the μMPS2 distribution. /USR/LOCAL/XXXX/UMPS2/ 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 the link commands is important: specifically, the first two support files must be in their respective positions.[2]

Finally, your OS can be tested by launching μMPS2. Entering:

```
UMPS2
```

without any parameters loads the file KERNEL.CORE.UMPS by default. See Chapter 9-*pops* for details on using the μMPS2 simulator, program loaded tape cartridges, and μMPS2 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 μMPS2 executable file.

[2] As documented in Section 8.1-*pops*, if one is working on a big-endian machine one should modify the above commands appropriately; substitute MIPS- for MIPSEL- above.

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

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

[7] 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).