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

Level 4 (the VM-I/O support level) of JaeOS 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 **CP15.R1.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 an appropriate SYS5 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 JaeOS's virtual memory support; i.e. the system Pager.

These exception handlers will run in kernel-mode with **CP15.R1.VMc**=1, while the U-proc's will run in user-mode with **CP15.R1.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 multi-programming 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 **CP15.R1.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 3.1. 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 Kseg0 and U-proc's run in user-mode with **CP15.R1.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 μARM's virtual memory support to give each U-proc a private address space mapped to Useg2 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, JaeOS provides for some pages to be shared by all U-proc's: the first n pages of Useg3. A malicious U-proc could therefore interfere with other U-proc's that were using Useg3 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, **CP15.R1.VMc**=0, Local Timer enabled, **SP** set to the penultimate RAM frame, and **PC**=`test`. The VM-I/O support level initialization function should therefore be called `test`.

3.1 SYS/Bp Exception Handling

As described in Section 3.3, the nucleus directly handles all SYS1-SYS10 SYSCALL exceptions and Breakpoint exceptions [1...4] (**LDST**, **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 specifying a handler 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 **a1**-**a4** immediately prior to executing a **SYSCALL** or **BREAK** instruction. The VM-I/O support level SYS/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 **a1**.

In particular, if a U-proc executes a **SYSCALL** instruction and **a1** contained a value in the range [11..20] then the VM-I/O support level should perform one of the services described below.

3.1.1 Read From Terminal (SYS11)

`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 SYS11 service is requested by the calling U-proc by placing the value 11 in **a1**, the virtual address of a string buffer where the data read should be placed in **a2**, and then executing a **SYSCALL** instruction. Once the process resumes the number of characters actually transmitted is returned in **a1** 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 **a1**.

Attempting to read from a terminal device to an address in Kseg0 is an error and should result in the U-proc being terminated (SYS20).

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

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

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

3.1.2 Write To Terminal (SYS12)

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 SYS12 service is requested by the calling U-proc by placing the value 12 in **a1**, the virtual address of the first character of the string to be transmitted in **a2**, the length of this string in **a3**, and then executing a **SYSCALL** instruction. Once the process resumes the number of characters actually transmitted is returned in **a1** 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 **a1**.

It is an error to write to a terminal device from an address in Kseg0, request a SYS12 with a length less than 0, or a length greater than 128. Any of these errors should result in the U-proc being terminated (SYS20).

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

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

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

3.1.3 V Virtual Semaphore (SYS13)

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

The V or SYS13 service is requested by the calling U-proc by placing the value 13 in **a1**, the *virtual* address of the semaphore to be V'ed in **a2**, the weight value (number of resources to be released) in **a3** and then executing a **SYSCALL** instruction.

Attempting to perform a V operation on an address in Kseg0 or Useg2 or to release a negative amount of resources are errors and should result in the U-proc being terminated (SYS20).

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

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

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

3.1.4 P Virtual Semaphore (SYS14)

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

The P or SYS14 service is requested by the calling U-proc by placing the value 14 in **a1**, the *virtual* address of the semaphore to be P'ed in **a2**, the wight value (number of resources to be allocated) in **a3** and then executing a **SYSCALL** instruction.

Attempting to perform a P operation on an address in Kseg0 or Useg2 or to alloc a negative amount of resources are errors and should result in the U-proc being terminated (SYS20).

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

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

Where the mnemonic constant `PSEMVIRT` has the value of 14.

3.1.5 Delay (SYS15)

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 SYS15 service is requested by the calling U-proc by placing the value 15 in **a1**, the number of seconds to be delayed in **a2**, 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 (SYS20).

The following C code can be used to request a SYS15:

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

Where the mnemonic constant `DELAY` has the value of 15.

3.1.6 Disk Put (SYS16) and Disk Get (SYS17)

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

The SYS16 (SYS17) service is requested by the calling U-proc by placing the value 16 (17) in **a1**, the virtual address of the 4KB block to be written to the disk (to contain the data from the disk) in **a2**, the disk number ([1...7]) in **a3**, the disk sector to be written onto (read from) in **a4**, and then executing a **SYSCALL** instruction. Once the process resumes **a1** 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 **a1**.

From one perspective disk devices are three dimensional devices: cylinders (or tracks), surfaces (or heads) and tracks (or sectors). From another perspective they are only one-dimensional: sectors. A disk device with x cylinders, y surfaces, and z tracks can be thought of being a (one dimensional) device with $sectCnt = x * y * z$ sectors numbered $[0...(sectCnt - 1)]$. The SYS16/SYS17 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 Kseg0 is an error and should result in the U-proc being terminated (SYS20). 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 (SYS20).

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

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

Where the mnemonic constant `DISK_PUT` has the value of 16. A SYS17 is requested by substituting `DISK_PUT` with `DISK_GET`, where the mnemonic constant `DISK_GET` has the value of 17.

3.1.7 Write To Printer (SYS18)

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 **a1**.

The SYS18 service is requested by the calling U-proc by placing the value 18 in **a1**, the virtual address of the first character of the string to be transmitted in **a2**, the length of this string in **a3**, and then executing a **SYSCALL** instruction. Once the process resumes the number of characters actually transmitted is returned in **a1** 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 **a1**.

It is an error to write to a printer device from an address in Kseg0, request a SYS18 with a length less than 0, or a length greater than 128. Any of these errors should result in the U-proc being terminated (SYS20).

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

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

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

3.1.8 Get TOD (SYS19)

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 **a1**.

The SYS19 service is requested by the calling U-proc by placing the value 19 in **a1** and then executing a **SYSCALL** instruction.

The following C code can be used to request a SYS19:

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

Where the mnemonic constant `GETTOD` has the value of 19.

3.1.9 Terminate (SYS20)

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

When all U-proc's have terminated, JaeOS should "shut down". Thus, somehow the "system" processes created in the VM-I/O support level (e.g. the delay daemon process see Section 3.3) 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 SYS20 service is requested by the calling process by placing the value 20 in **a1** and then executing a **SYSCALL** instruction.

The following C code can be used to request a SYS20:

```
void SYSCALL (TERMINATE)
```

Where the mnemonic constant `TERMINATE` has the value of 20.

3.2 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 SYS20 request.

3.3 Delay Facility

The SYS15 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.

3.3.1 Where to Store Sleeping U-proc's

As mentioned above, access to the nucleus is limited solely to requesting SYS1-SYS10 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 SYS15 as a request to perform a P operation on the U-proc's private semaphore.

3.3.2 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 list of delay-node descriptors whose head is pointed to by the list_head pointer `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 Linux kernel list.

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_list` list_head 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 **CP15.R1.VMc**=1 and all interrupts enabled.

3.4 Virtual P and V Service

The SYS13/SYS14 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 operation 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.

3.4.1 Where to Store Blocked U-proc's

When a U-proc performs a P (SYS14) 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.

3.4.2 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 SYS14 operation. The following implementation is suggested: maintain a list of virtSem-node descriptors whose head is pointed to by the list_head pointer `virtSemd_h`. The list `virtSemd_h` points to will represent the list of U-proc's blocked because of a SYS14 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 pointer `virtSemdFree_h`, is kept, like the delaydFree, pcbFree, and semdFree lists, as a Linux kernel list.

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_list` list_head field, 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 SYS14 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 SYS13 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 SYS14 on the same semaphore the order of unblocking/removal is first-in first-out.

3.5 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 with valid TLB exception handler, "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 was performed during U-proc initialization, execution continues with the VM-I/O support levels 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*/Chapter 3-*uarmdoc*. Which of these exception types are to be handled by the Pager and which are to trigger a SYS20 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.

3.5.1 System Segment Tables

As described in Section 3.2-*uarmdoc*, 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 Kseg0, Useg2, and Useg3 PTE's. These must be initialized during U-proc initialization.

All U-proc ASID's will share the same Kseg0 PTE; there is one Kseg0 PTE and all ASID Kseg0 PTE segment table pointers will point to it. The single Kseg0 PTE is a VM-I/O support level data structure. This segment table entry is only used when the U-proc generates a Kseg0 reference when in kernel-mode, since any reference to Kseg0 while in user-mode generates an Address Error exception.

All U-proc ASID's will share the same Useg3 PTE; there is one Useg3 PTE and all ASID Useg3 segment table pointers will point to it. The single Useg3 PTE is a VM-I/O support level data structure.

Each U-proc ASID will have its own Useg2 PTE. The array of Useg2 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 Kseg0 entry above and NULL for both the Useg2 and Useg3 entries.

3.5.2 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 Kseg0 and Useg3 segments. Exclusive of the Kseg0 PTE, each PTE should contain MAXPAGES entries (where MAXPAGES = 32).

For the Useg3 PTE, the MAXPAGES entries should describe the first MAXPAGES pages of the segment (0xC000.0000 0xC001.F000). Each page's PTE entry in **CP15.PTE_Hi** should indicate the **VPN** and **SEGNO** of the page, and the ASID should be set to zero. Each page's PTE entry in **CP15.PTE_Low** should indicate that the entry is global, but invalid (i.e. not present).

For the individual Useg2 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 **CP15.PTE_Hi** 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 **CP15.PTE_Low** should indicate that the entry is both not global and invalid.

As the above definitions illustrate, JaeOS 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 Useg3 is always within the segments first MAXPAGES pages.

The Layout of Kseg0 and its PTE

The installed physical RAM begins at 0x0000.7000 and goes up to RAMTOP. The default bootstrap loaders load the OS beginning at 0x0000.8000; the first frame of RAM is reserved for the ROM Reserved Frame. The OS **.text** and **.data** areas are loaded adjacent to each other starting at 0x0000.8000. 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 Useg2, 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 3.8.1.
- (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 1 illustrates one suggested organization.

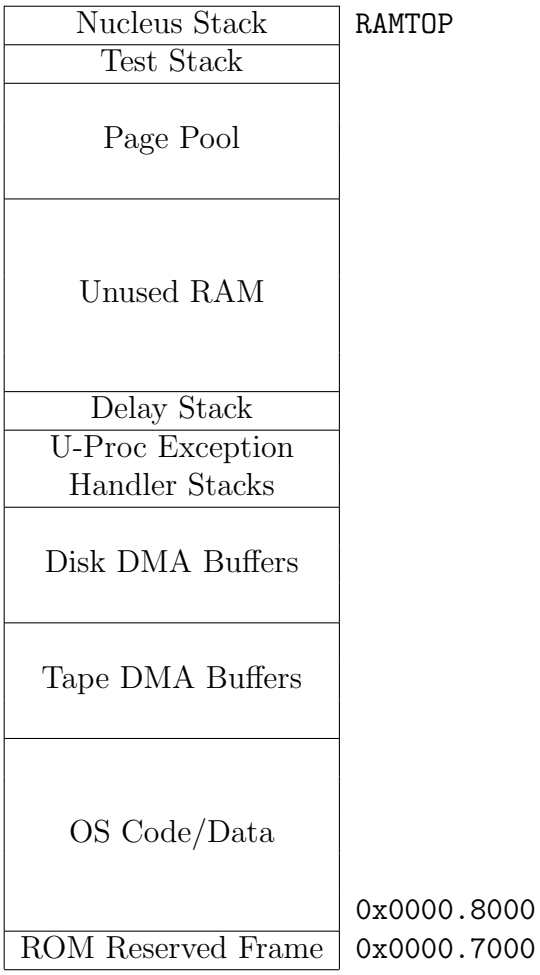

Figure 1: RAM Frame Layout Organization

The single Kseg0 PTE needs to be set up so that the VM-I/O support level exception handlers can run with **CP15.R1.VMc**=1, but that no page fault ever occurs for an address in Kseg0, and all addresses generated in Kseg0 get translated into the same physical address that would be generated if **CP15.R1.VMc**=0. The VM-I/O support level exception handlers need to run with **CP15.R1.VMc**=1 so that the U-proc's Useg2 (and Useg3) address space(es) are addressable. Furthermore, the VM-I/O support level exception handlers need to run as if **CP15.R1.VMc**=0 since these handlers interact with the nucleus and device registers which only understand physical addresses.

To accomplish this one should allocate a Kseg0 PTE of approximately 50 entries. These entries will describe the 50 physical frames beginning at `0x0000.7000`. Each page's PTE entry in **CP15.PTE_Hi** should indicate the **VPN** and **SEGNO** of the page, and the ASID should be set to zero. Each page's PTE entry in **CP15.PTE_Low** 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**=`0x00007` would have a **PFN**=`0x00007`.)

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 Kseg0 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 **CP15.R1.VMc**=0 it makes no use of the Kseg0 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.

3.5.3 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 preceding `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**).

3.5.4 Handling a TLB Invalid Exception

As described in Section 4.3.4-*pops*/Section 3.2-*uarmdoc* 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 SYS20 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 3.8.2.

3. Determine if the missing page is still missing. A page fault for a page in the shared segment (Useg3) 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*/Section 4.2-*uarmdoc* 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 μARM, 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.I**=1 & **Status.F**=1) 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 Useg3. How the backing store device's sectors are divided up among the U-proc's and Useg3 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 **CP15.PTE_Low** 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; μARM's re-attempt to translate a virtual address into a physical one.

3.5.5 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, μARM 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 **CP15.PTE_Low** 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 SYS20 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.

3.6 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 Kseg0 PTE.
2. Initialize the single Useg3 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 3.7.
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, SYS20, 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 a SYS2 to kill the Delay Daemon.
11. Invoke the SYS20 Terminate service to halt the OS.

3.7 U-proc Initialization

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

3.7.1 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 Useg2 PTE.
3. Initialize the U-proc's segment table.
4. Initialize the U-proc's private semaphore.

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 SYS5 request are yet another VM-I/O support level implemented data structure. Each New Area should be initialized to be a state with all interrupts enabled, user-mode off and **CP15.R1.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** to the address of the respective VM-I/O support level exception handler and **CP15.PTE_Hi.ASID** to the U-proc's assigned ASID.

6. Since it is imperative that the SYS5 request 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 all interrupts are enabled, user-mode is off and **CP15.R1.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** to the address of the U-proc step 2 initialization function and **CP15.PTE_Hi.ASID** to the U-proc's assigned ASID.

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

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

The second step in launching a U-proc is to issue the SYS5 request, 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 the ASID value.

2. Perform the SYS5 operation.

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 Useg2, 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 all interrupts are enabled, user-mode is on and **CP15.R1.VMc**=1. The **SP** should be set to the last page of Useg2. Finally set the **PC** to `0x8000.0054` (i.e. The contents of the second word in Useg2) and **CP15.PTE_Hi.ASID** to the U-proc's assigned ASID. (See Section 8.3-*pops* for an explanation as to why the **PC** are set to this value instead of just the beginning of Useg2.)

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 Useg2) and the second will be for the first page of code (the first page of Useg2). 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 Useg2.) 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.

3.8 Nuts and Bolts

3.8.1 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 *CP15* 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 3.5.2.) 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 **CP15.R1.VMc**=1 and its ASID set to the current U-proc. From one perspective the copy operation is from one virtual address (in Useg2 or Useg3) to another virtual address (in Kseg0). Any page faults that occur or the fact that the 4KB source block is not necessarily con-

tiguous in RAM is automatically taken care of by virtual address translation. From another perspective, given the way the Kseg0 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-proc initialization, the buffer for a tape device (filled via a tape read operation) can be used as the source for backing store disk write operation.

3.8.2 Managing Concurrency

Level 4 of JaeOS 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 3.5.4 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.

3.8.3 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 SYS13 (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.

3.8.4 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 3.7) 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 the ASID value.)

3.8.5 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.

3.8.6 Accessing the `libuarm` Library

As described in Chapter 4-*uarmdoc*, accessing the **CP15** registers and the ROM-implemented services/instructions in C is via the `libuarm` library. Simply include the line

```
#include ‘‘/usr/include/uarm/libuarm.h’’
```

The file libuarm.h is part of the μARM distribution.
/usr/include/uarm/ 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.

3.8.7 Testing

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

- helloTest.c - Hello World program for phase 3.
- 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.
- rogueTest.c - Tests the fault tolerance of the OS trying to do forbidden operations.
- tictactoe.c - Tic Tac Toe game, larger executable, tests .text section spanning on multiple pages.
- printerTest.c - Tests writing to the printer.
- fibTest.c - processor intensive job; calculate Fib(24) and Fact(12) recursively.
- readTest.c & multTest.c - Tests for correct terminal input.
- consumerTest.c & producerTest.c - Tests the VM-I/O support level P & V facility. These two programs must be run together.
- consDiskTest.c & prodDiskTest.c - Tests the VM-I/O support level P & V facility that also use Disk1 for synchronization, .data section is large and spans over multiple pages. These two programs must be run together.
- ulib.c - A utility module used by all of the above test programs to facilitate syscall requests and terminal printing.

Additionally, it is easy to write one's own programs to run under JaeOS. 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:

```
arm-none-eabi-gcc -pedantic -Wall -mcpu=arm7tdmi -c filename.c
```

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

```
arm-none-eabi-ld -T
   /usr/include/uarm/ldscripts/elf32ltsarm.h.uarmcore.x
   /usr/include/uarm/crtso.o /usr/include/uarm/libuarm.o
   phase1, phase2 & phase3 .o files -o kernel
```

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

```
elf2uarm -k kernel
```

which produces the file: KERNEL.CORE.UARM

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

```
arm-none-eabi-ld -T
   /usr/include/uarm/ldscripts/elf32ltsarm.h.uarmaout.x
   /usr/include/uarm/crti.o /usr/include/uarm/libuarm.o
   print.o swapTest.o -o swapTest
```

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

```
elf2uarm -a swapTest
```

which produces the file: SWAPTEST.AOUT.UARM
Finally, the linked file can be loaded onto a tape cartridge with the command:

```
uarm-mkdev -t swapTape.uarm swapTest.aout.uarm
```

which produces the file: SWAPTAPE.UARM

The files ELF32LTSARM.H.UARMCORE.X, LF32LTSARM.H.UARMAOUT.X, CRTSO.O, CRTI.O and LIBUARM.O are part of the μARM distribution. /USR/INCLUDE/UARM and /USR/INCLUDE/UARM/LDSCRIPTS 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.

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

```
uarm
```

without any parameters loads the file KERNEL.CORE.UARM by default. Use the settings panel inside the emulator to load tapes and disks.

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