

Guide to the AMIKaya Operating System Project

Enrico Cataldi

February 5, 2007

Contents

1. Introduction.....	3
2. Phase 1 – Level 2: The Queue Managers.....	4
2.1 ThreadBLK and MessageBLK Allocation/De-allocation.....	5
2.2 Thread Queue Maintenance.....	7
2.3 Thread Tree Maintenance.....	9
2.4 Message Queue Maintenance.....	10
2.5 Nuts and Bolts.....	12
2.5 Testing.....	13
3. Phase 2 – Level 3: The Nucleus.....	14
3.1 Nucleus Initialization.....	15
3.2 Scheduling.....	17
3.3 System Service Interface and Trap Management Threads.....	18
3.4 SYS/Bp Exception Handling.....	20
3.4.1 MsgSend.....	20
3.4.2 MsgRecv.....	21
3.4.3 Other SYSCALLs.....	21
3.5 PgmTrap Exception Handling.....	23
3.6 TLBTrap Exception Handling.....	24
3.7 Interrupt Exception Handling.....	25
3.8 Nucleus Services.....	26
3.8.1 CREATEBROTHER.....	26
3.8.2 CREATESON.....	26
3.8.3 TERMINATE.....	26
3.8.4 SPECPRGMGR.....	26
3.8.5 SPECTLBMGR.....	26
3.8.6 SPECSYSMGR.....	27
3.8.7 GETCPU TIME.....	27
3.8.8 WAITFORCLOCK.....	27
3.8.9 WAITFORIO.....	27
3.9 Nuts and Bolts.....	28
3.9.1 Timing Issues.....	28
3.9.2 Returning from a SYS/Bp Exception.....	28
3.9.3 Loading a New Processor State.....	28
3.9.4 Thread Termination Issues.....	29
3.9.5 Accessing the libumps Library.....	29
3.9 Testing.....	31
Further Development.....	32
Credits.....	32
References.....	33

1. Introduction

The AMIKaya Operating System (OS) described below is inspired to the previous experiences of Kaya OS[1] and AMIKE OS[2] projects. Both of them are descending (not directly) from the THE OS[3] outlined by E. Dijkstra. In his papers he described an OS divided into six layers. Each layer i provides an abstraction layer to the $i+1$ layer. Kaya derives from the past experiences of the TINA OS and MPS [6], a rework of the HOCA OS and CHIP [4,5]. The AMIKE specifications introduced a microkernel operating system, based on the message passing facility.

The OS we are going to describe is not complete as Dijkstra's one.

- Level 0: μ MPS hardware, described in the [μMPS Principles of Operation](#));
- Level 1: services provided in ROM (fully described in [μMPS Principles of Operation](#)):
 - processor state save/load
 - ROM-TLB-Refill handler
 - LDST, FORK, PANIC, HALT
- Level 2: the Queues Managers (Phase 1 – described in Chapter 2). Based on the key operating system concept that active entities at one layer are just data structures at lower layers, this layer support management of queues of structures; ThreadBLKs and MessageBLKs
- Level 3: the Nucleus (Phase 2 – described in Chapter 3). This level implements the thread scheduling, interrupt handling, message passing, deadlock detection, System Service Interface and its services

The AMIKaya project actually provides a source code that implements all these levels.

Further levels could be developed:

- Level 4: the Support Level. The 3rd level is extended to a system that can support multiple user-level cooperating threads that can request I/O and which run on their own virtual address space. Furthermore, this level adds user-level synchronization, message passing and a thread sleep/delay facility
- Level 5: the Network Level. This level implements a minimal TCP/IP stack to get provide access with existing virtual Ethernet devices to a real network, through VDE[7]
- Level 6: the File System. This level implements the abstraction of a flat file system with primitives to create, rename, delete, open, close and modify files
- Level 7: the Interactive Shell

2. Phase 1 – Level 2: The Queue Managers

Level 2 of AMIKaya 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 threads and messages, and what represent them at this level are thread control blocks (ThreadBLKs) and message blocks (MessageBLKs).

```
/* thread control block */
typedef struct tcb_t {
    /* thread queue fields */
    struct tcb_t *t_next, /* pointer to next entry
*/
    /* thread tree fields */
    *t_parent, /* pointer to parent */
    *t_sibling, /* pointer to next
sibling */
    *t_child; /* pointer to 1st child
*/
    /* thread's message queue */
    struct msg_t *inbox;
    struct state_t proc_state; /* processor state */
    /* Other fields will be added during phase2 development
*/
} tcb_t;
```

The Thread Queue Manager will implement functions to provide these services:

- Allocation/de-allocation of single ThreadBLK elements
- Maintenance of ThreadBLK queues
- Maintenance of ThreadBLK trees

```
/* message block */
typedef struct msg_t{
    struct msg_t *m_next; /* pointer to next entry */
    struct tcb_t *m_sender; /* thread that sent this message
*/
    unsigned int message; /* the payload of the message */
} msg_t;
```

The Message Queue Manager will implement functions to provide these services:

- Allocation/de-allocation of single MessageBLK elements
- Maintenance of MessageBLK's queues

2.1 ThreadBLK and MessageBLK Allocation/De-allocation

One may assume that AMIKaya supports no more than MAXTHREADS concurrent threads and no more than MAXMESSAGES messages in transit; these parameters should be set to 20 (const.h).

Thus, this level needs a number of MAXTHREADS ThreadBLKs and MAXMESSAGES MessageBLKs to allocate from and de-allocate to.

Assuming that there is a set of MAXTHREADS ThreadBLKs and a set of MAXMESSAGES MessageBLKs, those unused may be kept on a circular, singly linked, tail pointed list: one for ThreadBLKs and one for MessageBLKs.

Entity allocation and de-allocation will be provided by these procedures:

void initTcbs(void);

Initialize the tcbFree list.

This method will be called only once during data structure initialization

void freeTcb(tcb_t *t);

Insert the element pointed to by t onto the tcbFree list

tcb_t * allocTcb(void);

Return NULL if the tcbFree list is empty, otherwise remove an element from the tcbFree list, provide initial values for *all* of the ThreadBLK's fields (i.e. NULL and/or 0) and then return a pointer to the removed element; ThreadBLKs get reused, so it is important that no previous value remains in a ThreadBLK when it gets reallocated

void initMsgs(void);

Initialize the msgFree list.

This method will be called only once during data structure initialization

void freeMsg(msg_t *m);

Insert the element pointed to by m onto the msgFree list

msg_t *allocMsg(void);

Return NULL if the msgFree list is empty, otherwise remove an element from the msgFree list, provide initial values for *all* of the MessageBLK's fields (i.e. NULL and/or 0) and then return a pointer to the removed element; MessageBLKs get reused, so it is important that no previous value remains in a MessageBLK when it gets reallocated

Since AMIKaya will run without any dynamic memory allocation/deallocation facility, runtime

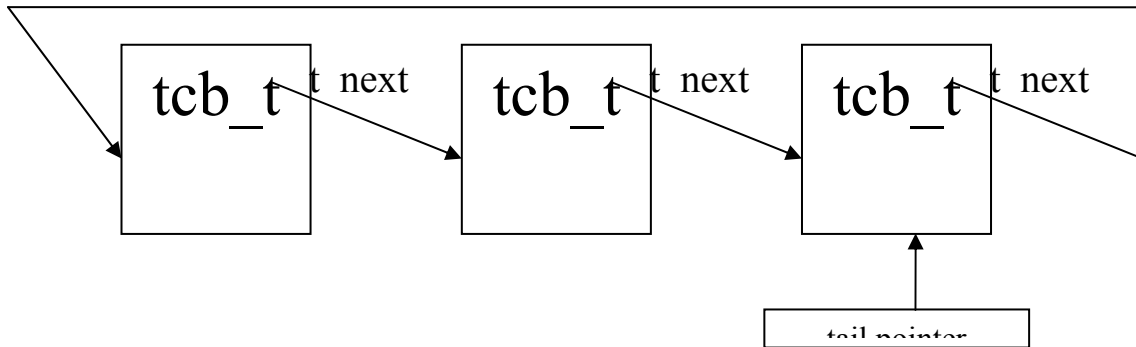
resource request is not allowed, the best way to solve this problem is to have a initial allocation of all the required space, whose lifetime would be exactly the same of the kernel's one.

This can be done during initialization of data structures, using a statical allocation of two arrays, one for ThreadBLKs and one for MessageBLKs, of the requested size (MAXTHREADS and MAXMESSAGES).

2.2 Thread Queue Maintenance

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

Queues to be manipulated are circular, singly linked and tail pointed lists. One may optionally make them doubly linked for greater efficiency.



To provide support to thread queues, the following externally visible functions should be implemented:

```
tcb_t * mkEmptyThreadQ(void);
```

Used to initialize a variable to be tail pointer to a thread queue; returns a pointer to the tail of an empty thread queue, i.e. NULL

```
int emptyThreadQ(tcb_t *tp);
```

Returns TRUE if the queue whose tail is pointed to by tp is empty, FALSE otherwise.

```
void insertThread(tcb_t **tp, tcb_t *t_ptr);
```

Insert the ThreadBLK pointed to by t_ptr into the thread 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

```
tcb_t * removeThread(tcb_t **tp);
```

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

```
tcb_t * outThread(tcb_t **tp, tcb_t *t_ptr);
```

Remove the ThreadBLK pointed to by t_ptr from the queue whose tail-pointer is pointed to by tp. Update the queue's tail pointer if necessary. If the desired entry is not in the queue (an error condition), return NULL; otherwise, return t_ptr. Note: t_ptr can point to any element of the queue

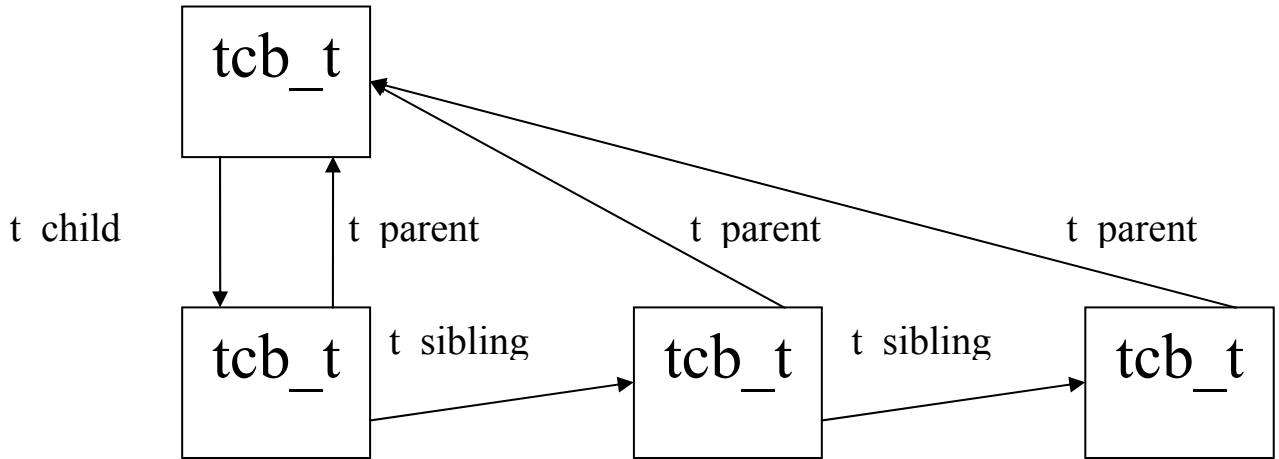
```
tcb_t * headThread(tcb_t *tp);
```

Return a pointer to the first ThreadBLK from the queue whose tail is pointed to by tp. Do not remove the ThreadBLK from the queue. Return NULL if the queue is empty

2.3 Thread Tree Maintenance

In addition to possibly participating in a thread queue, ThreadBLKs are also organized into trees of ThreadBLKs, called thread trees. Each ThreadBLK provides a pointer to its parent, its children and its siblings.

One may optionally make the links doubly linked at sibling level for greater efficiency.



To support thread trees, the following externally visible functions should be implemented:

```
int emptyChild(tcb_t *t);
```

Return TRUE if the ThreadBLK pointed to by t has no children, FALSE otherwise.

```
void insertChild(tcb_t *prnt, tcb_t *t);
```

Make the ThreadBLK pointed to by t a child of the ThreadBLK pointed to by prnt. In other words: insert t in the thread tree as a child of prnt

```
tcb_t * removeChild(tcb_t *t);
```

Make the first child of the ThreadBLK pointed by t no longer a child of t. Return NULL if there are no t's children, otherwise a pointer to the removed ThreadBLK first child. In other words: if t has children, remove the first one from the tree and return a pointer to it, otherwise return NULL

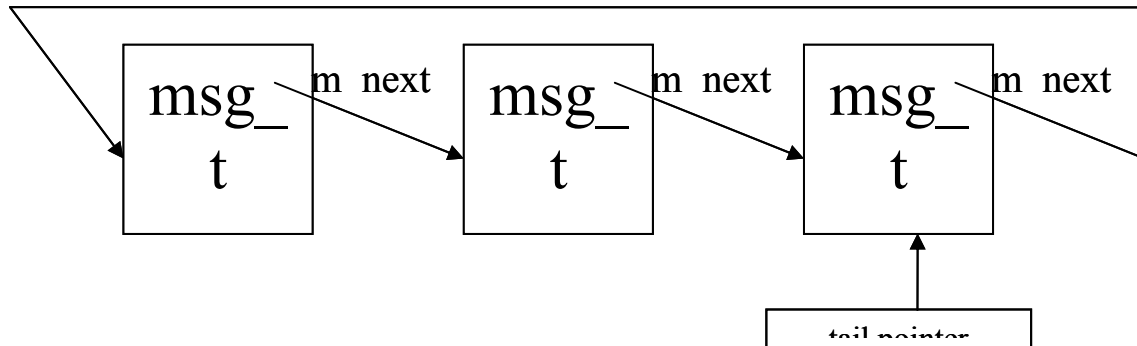
```
tcb_t * outChild(tcb_t *t);
```

Make the ThreadBLK pointed to by t no longer the child of its parent. If the ThreadBLK pointed to by t has no parent, return NULL, otherwise return t. The element pointed by t need not be the first child of its parent. In other words: look in the tree for the ThreadBLK pointed by t; if t has no parent, return NULL; if it does have a parent, remove t from the tree and return t

2.4 Message Queue Maintenance

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

Queues to be manipulated are circular, singly linked and tail pointed lists. One may optionally make them doubly linked for greater efficiency.



To provide support to message queues, the following externally visible functions should be implemented:

```
msg_t * mkEmptyMessageQ(void);
```

Used to initialize a variable to be tail pointer to a message queue; returns a pointer to the tail of an empty message queue, i.e. NULL

```
int emptyMessageQ(msg_t *mp);
```

Returns TRUE if the queue whose tail is pointed to by mp is empty, FALSE otherwise

```
void insertMessage(msg_t **mp, msg_t *m_ptr);
```

Insert the MessageBLK pointed to by m_ptr at the **end** of the queue whose tail-pointer is pointed to by mp; note the double indirection through mp to allow for the possible updating of the tail pointer as well

```
void pushMessage(msg_t **mp, msg_t *m_ptr);
```

Insert the MessageBLK pointed to by m_ptr at the **head** of the queue whose tail-pointer is pointed to by mp; note the double indirection through mp to allow for the possible updating of the tail pointer as well

```
msg_t * popMessage(msg_t **mp, tcb_t *t_ptr);
```

Remove the first element (starting by the head) from the message queue accessed via mp whose sender is t_ptr. If t_ptr is NULL,

return the first message in the queue. Return NULL if the message queue was empty or if no message from t_ptr was found; otherwise return the pointer to the removed message. Update the message queue's tail pointer if necessary

msg_t * headMessage(msg_t *mp);

Return a pointer to the first MessageBLK from the queue whose tail is pointed to by mp. Do not remove the MessageBLK from the queue. Return NULL if the queue is empty

2.5 Nuts and Bolts

There isn't just one way to implement the functionality of this level.

One approach is to create two modules: one for the Thread initialization/allocation/de-allocation, queue maintenance, and process tree maintenance (eg. `tcb.c`), and one for Message initialization/allocation/de-allocation and queue maintenance (eg. `msg.c`).

Another approach is to create a generic queue manager module (eg. `queue.c`) and then use it to build the bulk of message and thread queue manager modules, then adding the ADT-specific functions to each.

Regarding optimization, efficiency may be improved by introducing doubly-linked queues, adding eg. `t_prev` to ThreadBLKs, `m_prev` to MessageBLKs, and `t_prev_sibling` for ThreadBLKs tree management.

Each module should export its public interface using a `.e` file.

As with any non-trivial system, you are strongly encouraged to use the *make* program to maintain your code.

Initial structure initialization can be accomplished by statical allocation of two arrays, one for ThreadBLKs and one for MessageBLKs, of the requested size (`MAXTHREADS` and `MAXMESSAGES`), eg.:

```
HIDDEN tcb_t tcbTable[MAXTHREADS];  
HIDDEN msg_t msgTable[MAXMESSAGES];
```

The elements in these arrays will be collected into free lists, which could be managed as tail-pointed lists or queues, by using static pointers eg.:

```
HIDDEN tcb_t *tcbFree_h;  
HIDDEN msg_t *msgFree_h;
```

The insertion of array elements can be done during the initialization of data structures.

2.5 Testing

There is a provided test file, `pltest.c` that will “exercise” your code.

You should compile the source files separately using the commands eg.:

```
mipsel-linux-gcc -ansi -pedantic -Wall -c tcb.c
mipsel-linux-gcc -ansi -pedantic -Wall -c msg.c
mipsel-linux-gcc -ansi -pedantic -Wall -c pltest.c
```

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

```
mipsel-linux-ld -T $SUPDIR/elf32ltsmip.h.umpscore.x
$SUPDIR/crtso.o $SUPDIR/libumps.o tcb.o msg.o pltest.o -o kernel
```

Where `$SUPDIR` must be replaced with the path to the μ MPS support directory and `elf32ltsmip.h.umpscore.x`, `crtso.o` and `libumps.o` are part of the μ MPS distribution.

If one is working on a big-endian machine one should modify the above commands appropriately; substitute `mips-` for `mipsel-` and `elf32btmip.h.umpscore.x` for `elf32ltsmip.h.umpscore.x`.

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

```
umps-elf2umps -k kernel
```

which produces the file `kernel.core.umps`

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

```
umps
```

without any parameters loads the file `kernel.core.umps` by default.

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

3. Phase 2 – Level 3: The Nucleus

Level 3 of AMIKaya builds on previous levels in two key ways:

- Building on the exception handling facility of Level 1 (the ROM-Excpt handler). 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 thread scheduler, message passing facility, System Service Interface and other minor services.

The purpose of the nucleus is to provide an environment in which asynchronous sequential threads exist, each making forward progress as they take turns sharing the CPU. Furthermore, the nucleus provides these threads with exception handling routines, message passing primitives. Last but not least, the core provides functionality specified in the SSI (*System Service Interface*).

SSI provides mechanisms to allow threads to proliferate, to synchronize with devices (I/O or Pseudo-Clock), and to provide them the ability to specify the thread that must be used as their trap handler. Trap that can be raised and handled may be Pgm, TLB and certain SYS/Bp exceptions.

Since virtual memory is not supported by AMIKaya Level 3, all addresses at this level are assumed to be physical addresses. Nevertheless, the nucleus needs to preserve the state of each thread. If a thread 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 Nucleus Initialization

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

- Populate the four New Areas in the ROM Reserved Frame.

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, and be in kernel-mode.

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

```
initTcbs();
initMsgs();
```

- Initialize all nucleus maintained variables (see further on)
- Instantiate a single thread for the SSI and place its ThreadBLK in the Ready Queue (see further on). A thread is instantiated by allocating a ThreadBLK (i.e. `allocTcb()`), and initializing the processor state that is part of it. In particular this thread may have interrupts enabled or disabled (see further on), virtual memory off, kernel-mode on, **\$SP** set to `RAMTOP-FRAMESIZE` (i.e. use the penultimate RAM frame for its stack), and its **PC** set to the address of SSI function entry point. One can assign a variable (i.e. the **PC**) the address of a function by using

```
... = (memaddr) SSI_function_entry_point;
```

where `memaddr`, in `TYPES.H` has been aliased to `unsigned int`. Remember to declare the SSI function entry point as “external” in your program by including the line:

```
extern void SSI_function_entry_point();
```

Remember that for technical reasons, whenever one assigns a value to the **PC** one must also assign the same value to the general purpose register **t9**. (a.k.a. `s_t9` as defined in `TYPES.H`.) Hence this will be done when initializing the four New Areas as well as the processor state that defines this single thread.

- Instantiate a single thread for `test` and place its ThreadBLK in the Ready Queue. Test is a supplied function/thread that will help you debug your nucleus. The same rules described for SSI apply here. `test` should start with interrupts enabled, virtual memory off, kernel-mode on, **\$SP** set to `RAMTOP-(2*FRAMESIZE)` (i.e. use the second-to-last RAM frame for its stack), and its **PC** set to the `test` address. Remember that test will require quite a large stack space to instantiate all its children threads.
- 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 threads 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 kernel-mode with virtual memory disabled and all interrupts masked. 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.

3.2 Scheduling

Your nucleus should guarantee finite progress; consequently, every ready thread 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:

- *Current Thread*: the thread that has currently the control of the CPU
- *Ready Queue*: the queue which identifies all threads that are awaiting for the CPU control
- *Thread Count*: the number of threads in the system
- *Soft-Block Count*: the number of threads that are blocked awaiting for I/O or completion of a service request by the SSI

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

- if the thread count = 1 and the SSI is the only thread in the system: perform *normal system shutdown*, that is, calling the HALT ROM routine
- if a deadlock is detected (that is, after system boot, if the thread count is higher than zero but the soft-block count is zero): perform *emergency shutdown*, that is, calling the PANIC ROM routine
- if thread count and soft-block count are higher than zero, some threads are waiting for I/O to complete: the system should enter in a *wait state* (that is, waiting until some device completes its operation). This is typically implemented by the execution of an infinite loop with interrupts enabled.

3.3 System Service Interface and Trap Management Threads

The System Service Interface (SSI), according to the AMIKaya OS Project specifications, is a fundamental component of the kernel, since it provides services which are needed to build up higher levels of AMIKaya, such as thread synchronization with I/O operation completion, *pseudo-clock tick* management, thread proliferation (creation of childs and brothers of threads), thread termination and trap management threads specifications.

The SSI runs inside the Kernel address space; ach relevant system event will become a message managed by the SSI.

The SSI will manage thread requests on behalf of the nucleus; for example, a thread which requires to know its accounted CPU time will send a message to the SSI, and wait for the answer.

If the SSI ever gets terminated, the system must be stopped performing an emergency shutdown.

The SSI thread should implement the following RPC server algorithm:

```
while (TRUE) {
    receive a request;
    satisfy the received request;
    send back the results;
}
```

Services that the SSI provides are:

- Fork: brother creation
Thread Fork Facility: the new thread will be inserted into parent's genealogy as its brother.
- Fork: child creation
Thread Fork Facility: the new thread will be inserted into parent's genealogy as its child.
- Thread Termination Request
Thread Termination Facility: this will terminate the thread and all its progeny.
- PRG or TLB or SYS Trap Manager Specification
Thread Trap Manager Specification Facilities: this will let specify an appropriate Trap Management Thread for each of these exceptions.
- Get CPU Time
Provides the total CPU Time that has been used from the requesting thread since its creation.
- Freeze until next pseudo-clock tick
Freezes requesting thread until the next pseudo-clock tick event.
- Freeze until I/O operation completion
Freezes requesting thread until I/O completion.

A thread makes a service request by using a specific function:

```
void SSIRequest(unsigned int service, unsigned int payload,
unsigned int * reply)
```

where *service* is a mnemonic code identifying the service requested, *payload* contains an argument (if required) for the service, and *reply* will point to the area where the answer (if required) should be stored.

If *service* does not match any of those provided by the SSI, the SSI should terminate the thread and its progeny. Also, when a thread requires a service to the SSI, it *must wait* for the answer.

While `SSIREquest` has to be implemented depending on specific SSI implementation, these general issues have to be addressed:

- SSI requests should be implemented using SYSCALLs and message passing
- there should be a way to identify requests addressed to SSI from other messages; the easiest way is to make the SSI thread address available to requestor threads, but this could lead to a kernel security leak, so a *magic number* (a “fake” address recognized by the nucleus) could be used to address requests to SSI. This magic number must be chosen carefully, to avoid addressing ambiguities and/or address clashing with other threads
- SSI requests and answers could require more than one parameter; `msg_t` could be used in a creative way and/or expanded to allow the transport of “fat” messages

Beside the SSI, in a complete kernel other threads will perform specific functions on behalf of the nucleus; in AMIKaya, these will be named *Trap Management Threads*.

The SSI is the only server thread which you must write in AMIKaya Phase 2: some sample trap management threads will be provided for testing purposes (the real implementation is demanded to AMIKaya's successive phases).

Trap Management Threads will run inside the Kernel address space, and will not exchange messages directly with other threads.

Each Management Thread implements the following RPC server algorithm:

```
while (TRUE) {
    receive a trap management request;
    manage the request;
    send back a decision upon continuing the trapped thread or
not;
}
```

Upon exception, the current thread is stopped and its state is saved in the old area; the nucleus should update the thread processor state with the one saved in the old area, and dispatch a message to the trap management thread with the current thread as sender, and the value of the **CAUSE** register as payload.

The trap manager will act upon the request and, at last, will send a message to the thread raising the trap with `TRAPCONTINUE` or `TRAPTERMINATE` as payload; the nucleus must then intercept this message and cause the thread termination or continuation.

3.4 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 thread 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 thread executing the SYSCALL or BREAK instruction depending on the value found in a0. In particular, if the thread making a SYSCALL request was in **kernel-mode** and **a0** contained 1 or 2 then the nucleus should perform a message passing related action.

3.4.1 MsgSend

This system call cause the transmission of a message to a specified thread. This is an asynchronous operation (sender doesn't wait for receiver to perform a SYS2) and on success returns zero, otherwise MSGNOGOOD is used to provide a meaningful error condition on return. This system call doesn't require the thread to lose its remaining time slice (it depends on scheduler policy). If the target thread is in the ready queue, this message is just pushed in its *inbox*, otherwise if the thread is awaiting for a message, then it has to be awakened and put into the Ready Queue.

A request will be well formed if:

- **a0** contains 1
- **a1** contains the *destination thread ThreadBLK address*
- **a2** contains the *payload* of the message

By setting code = SEND = 1, the following C macro can be used to request a SYS1:

```
#define MsgSend(code,dest,payload)
(SYSCALL((unsigned int) (code),(unsigned int)
(dest),(unsigned int) (payload),0))
```

or a wrapper function can be used:

```
unsigned int MsgSend(unsigned int code, tcb_t * dest,
unsigned int payload)
{
    unsigned int retcode;
    retcode = SYSCALL(code, (unsigned int) dest, payload, 0);
    return retcode;
}
```

3.4.2 *MsgRecv*

This system call is used by a thread to extract a message from its *inbox* or, if this one is empty, to wait for a message. This is a synchronous operation since the requesting thread will be frozen until a message matching the required characteristics doesn't arrive. This system call provides as returning value the identifier of the thread which sent the message extracted. This system call may cause the thread to lose its remaining time slice, since if its *inbox* is empty it has to be frozen.

A request will be well formed if:

- **a0** contains 2
- **a1** contains the *sender thread* or *ANYMESSAGE*
- **a2** contains a *pointer* to an area where the nucleus will store the *payload* of the message

If **a1** contains a *ANYMESSAGE* pointer, then the requesting thread is looking for the first message in its *inbox*, without any restriction about the sender. In this case it will be frozen only if the queue is empty, and the first message sent to it will wake up it and put it in the Ready Queue.

By setting `code = RECV = 2`, the following C macro can be used to request a SYS2:

```
#define MsgRecv(code,source,reply) (((tcb_t *)  
SYSCALL((unsigned int) (code),(unsigned int)  
(source),(unsigned int) (reply),0)))
```

or a wrapper function can be used:

```
tcb_t * MsgRecv(unsigned int code, tcb_t * source, unsigned  
int * reply)  
{  
    tcb_t * sender;  
    sender = (tcb_t *) SYSCALL(code, (unsigned int) source,  
(unsigned int) reply, 0);  
    return sender;  
}
```

3.4.3 *Other SYSCALLs*

Otherwise, if the value of **a0** is different from 1 or 2, or the request was made from a thread in **user-mode**, then the exception must be managed in another way.

The nucleus's SYSTrap exception handler will take one of two actions depending on whether the offending (i.e. current) thread has specified a SYSTrap Management Thread:

- if the offending thread hasn't specified a thread to manage a SYSTrap Exception, then the current thread and all of its progeny must be terminated.
- If the offending thread has specified a thread to manage a SYSTrap Exception, then the thread must be frozen, a message to the SYSTrap Management Thread must be sent with the cause of the exception as payload, and the offending thread as sender.
- The Management Thread then must decide where the execution of the thread must be terminated or can continue.

- The Management Thread will send a message to the offending thread with TRAPCONTINUE or TRAPTERMINATE as payload; the nucleus must intercept this message and cause the thread termination or continuation.

3.5 PgmTrap Exception Handling

A PgmTrap exception occurs when the executing thread attempts to perform some illegal or undefined action.

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) thread has specified a PgmTrap Management Thread:

- If the offending thread hasn't specified a thread to manage a PgmTrap Exception, then the current thread and all of its progeny must be terminated.
- If the offending thread has specified a thread to manage a PgmTrap Exception, then the thread must be frozen, a message to the PgmTrap Management Thread must be sent with the cause of the exception as payload and the offending thread as sender.
- The Management Thread then must decide where the execution of the thread must be terminated or can continue.
- The Management Thread will send a message to the offending thread with TRAPCONTINUE or TRAPTERMINATE as payload; the nucleus must intercept this message and cause the thread termination or continuation.

3.6 TLBTrap Exception Handling

A TLBTrap exception occurs when μ MPS fails in an attempt to translate a virtual address into its corresponding physical address.

Assuming that the TLBTrap 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 TLBTrap exception is raised, execution continues with the nucleus's TLBTrap exception handler.

The cause of the TLBTrap exception will be set in Cause.ExcCode in the TLBTrap Old Area.

The nucleus's TLBTrap exception handler will take one of two actions depending on whether the offending (i.e. Current) thread has specified a TLBTrap Management Thread:

- If the offending thread hasn't specified a thread to manage a TLBTrap Exception, then the current thread and all of its progeny must be terminated.
- If the offending thread has specified a thread to manage a TLBTrap Exception, then the thread must be frozen, a message to the TLBTrap Management Thread must be sent with the cause of the exception as payload and the offending thread as sender.
- The Management Thread then must decide where the execution of the thread must be terminated or can continue.
- The Management Thread will send a message to the offending thread with TRAPCONTINUE or TRAPTERMINATE as payload; the nucleus must intercept this message and cause the thread termination or continuation.

3.7 Interrupt Exception Handling

A device interrupt occurs when either a previously initiated I/O request completes or when the Interval Timer makes a 0x0000.0000 → FFFF.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-Excp 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. Furthermore, for interrupt lines 3–7 the Interrupting Devices Bit Map will indicate which devices on each of these interrupt lines have a pending interrupt.

Since AMIKaya will not generate software interrupts, interrupt lines 0 and 1 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.

Terminal devices are actually two sub-devices; a transmitter and a receiver. These two sub-devices operate independently and concurrently. Both sub-devices may have an interrupt pending simultaneously.

For purposes of prioritizing pending interrupts, terminal transmission (i.e. writing to the terminal) is of higher priority than terminal receipt (i.e. reading from the terminal). Hence the Interval Timer is the highest priority interrupt and reading from terminal 7 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 Interval 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 the Interval Timer is acknowledged by loading the Interval Timer with a new value.
- If the interrupt is raised from an I/O device this will be translated into a message for the SSI. The sender of message will be the device register base address and the payload will be the STATUS register value of the device raising the interrupt exception. This information may have to be stored by the SSI.
- If the interrupt was caused from the Interval Timer, the reason could be:
 - the current thread time-slice has expired: in this case, the scheduler must take the necessary actions to give the CPU access to another thread
 - a *pseudo-clock tick* has occurred: the nucleus should send the SSI a message to let the SSI unblock all threads who requested a WAITFORCLOCK service.

In both cases, CPU time accounting should be performed by the nucleus on behalf of threads.

3.8 Nucleus Services

These services should be implemented by the SSI on behalf of the nucleus, and could be requested by threads running in Kernel mode.

These services will be identified by mnemonic values.

3.8.1 CREATEBROTHER

This service should allow the requestor to create a new thread whose initial processor state is passed by reference as payload. The new thread should be inserted in the thread tree as a brother of the requestor. If the creation went fine, this service should return the created thread address, otherwise should return CREATENOGOOD.

3.8.2 CREATESON

This service should allow the requestor to create a new thread whose initial processor state is passed by reference as payload. The new thread should be inserted in the thread tree as a son of the requestor. The new thread should inherit trap manager definitions from the parent, if defined. If the creation went fine, this service should return the created thread address, otherwise should return CREATENOGOOD.

3.8.3 TERMINATE

This service should allow the requestor to terminate itself and its progeny recursively. Messages sent from the requestor or from one of its progeny to other threads still have to be delivered, but the SSI and/or the nucleus should discard further requests from the terminated threads.

There are no return codes.

3.8.4 SPECPRGMGR

This service should allow the requestor to specify his own trap management thread for Program Trap exceptions. The trap manager thread address is passed as payload; if the management thread does not exist, the requestor thread and its progeny should be terminated.

Once assigned or inherited, the trap manager thread cannot be redefined: further service requests of this type should cause the termination of the thread and its progeny.

3.8.5 SPECTLBMGR

This service should allow the requestor to specify his own trap management thread for TLB Management exceptions. The trap manager thread address is passed as payload; if the management thread does not exist, the requestor thread and its progeny should be terminated.

Once assigned or inherited, the trap manager thread cannot be redefined: further service requests of this type should cause the termination of the thread and its progeny.

3.8.6 SPECSYSMGR

This service should allow the requestor to specify his own trap management thread for SYSCALL and Break exceptions. The trap manager thread address is passed as payload; if the management thread does not exist, the requestor thread and its progeny should be terminated.

This is the trap management thread that will be called when “pass up” of SYSCALLs and service requests is attempted.

Once assigned or inherited, the trap manager thread cannot be redefined: further service requests of this type should cause the termination of the thread and its progeny.

3.8.7 GETCPU TIME

This service should allow the requestor get back the CPU time used up until now (in microseconds).

The scheduler policy should define how SYSCALLs, interrupts and other nucleus’ activities on behalf of threads should be accounted.

3.8.8 WAITFORCLOCK

One of the services the nucleus has to implement is the *pseudo-clock*, that is, a virtual device which sends out an interrupt (a *tick*) every 100 milliseconds. This interrupt will be translated into a message to the SSI, as for other interrupts.

WAITFORCLOCK should allow the requestor to suspend its execution until the next pseudo-clock tick. Pseudo-clock management should be quite precise: at most one of such messages should wait in the SSI *inbox*. This could be obtained by prioritizing the message.

3.8.9 WAITFORIO

Threads may start I/O operations by writing commands in device registers.

To avoid busy waiting, WAITFORIO should allow threads to request to be suspended until the I/O operation is completed.

The device is identified by the thread by specifying its *device register base address* (that is, the address of its **STATUS** register) as payload.

If the device does not exist, the thread and its progeny should be terminated.

Upon I/O operation completion, the **STATUS** register value of the device should be returned to the requestor thread.

Because of CPU scheduling, the I/O operation could be completed before the service request could be processed by the SSI: the **STATUS** upon I/O completion has to be recorded until the thread performs the service request.

3.9 Nuts and Bolts

3.9.1 Timing Issues

While μ MPS has two clocks, the TOD clock and Interval Timer, only the Interval Timer can generate interrupts. Hence the Interval Timer must be used simultaneously for two purposes: generating interrupts to signal the end of threads time slices, and to signal the end of each 100 millisecond period (a pseudo-clock tick). The nucleus should be able to load the most appropriate value into the Interval Timer to manage both requirements, and to tell if the Interval Timer interrupt has happened because of pseudo-clock tick or time slice expiration, and act accordingly.

The CPU time used by each thread must also be kept track of (GETCPU TIME). This could be implemented by adding a field to the ThreadBLK structure. While the Interval Timer is useful for generating interrupts, the TOD clock is useful for recording the length of an interval. By storing off the TOD clock's value at both the start and end of an interval, one can compute the duration of that interval.

The Interval Timer and TOD clock are mechanisms for implementing AMIKaya's scheduler policy. Scheduler 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 thread, 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 thread.
- While the time spent by the nucleus handling a SYSCALL request needs to be measured for pseudo-clock tick purposes, which thread, if any, should be "charged" with this time.

It is also necessary to decide if manage the CPU time accounting using only a 32-bit counter (easier to implement, but introduces limits on maximum CPU time values) or a 64-bit counter, which has no limitations could be split in two like TODHI/TODLO, but requires 64-bit algebra to be devised.

Another important issue to be managed is the conversion of CPU ticks into microseconds or milliseconds: there is a Timescale register which allows such a conversion to be made.

3.9.2 Returning from a SYS/Bp Exception

SYSCALLs that do not result in thread termination or suspension could return control to the requesting threads execution stream: this depends on scheduler policy.

In any case, when the thread has to be restarted, 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 thread will result in an infinite loop of SYSCALL's. To avoid this the PC must be incremented by 4 (i.e. the μ MPS 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.

3.9.3 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. Whenever the ROM-Excpt handler loads a processor state a pop operation is performed on the KU/IE and VM stacks. 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 & KUp) for the desired assignment to take effect after the ROM-Excpt handler loads the processor state.

3.9.4 Thread Termination Issues

When a thread is terminated there is actually a whole (sub)tree of threads that get terminated. There are a number of tasks that must be accomplished:

- The root of the sub-tree of terminated threads must be “orphaned” from its parents; its parent can no longer have this ThreadBLK as one of its progeny.
- Future messages directed to terminated thread must produce an error returning condition.
- Message sent from the terminated thread before its termination must not be removed, but pending service requests have to be dropped by the SSI and/or the nucleus.
- The thread count and other nucleus internal variables need to be adjusted accordingly.
- Threads (i.e. ThreadBLK's) can't hide. A ThreadBLK is either the current thread, sitting on the ready queue, or is blocked somewhere.

Termination may be accomplished by the SSI (TERMINATE service) and by the nucleus; it could be easier to develop just one general termination function (eg. belonging to the nucleus) and have both SSI and nucleus use it.

3.9.5 Data Structures for I/O Management

The SSI has to match requests made by threads (WAITFORCLOCK, WAITFORIO) with events from devices, that is, messages notifying device interrupts and pseudo-clock ticks.

Requests and events may arrive in any order, so the SSI has to keep track of threads waiting, device STATUSes, and so on. All this data has to be collected and managed by the SSI, and could be implemented by adding fields to `tc_b_t` data structure and/or by adding whole new data structures to the SSI, eg. an I/O waiting thread queue, a device STATUS array, and so on.

It also depends also on scheduler implementation; eg. if the scheduler does have a I/O wait queue, it could be used to handle threads waiting on I/O on behalf of SSI.

3.9.6 Accessing the libumps Library

Accessing the CP0 registers and the ROM-implemented services/instructions in C is via the libumps library. Simply include the line:

```
#include "$SUPDIR/e/libumps.e"
```

The file libumps.e is part of the μ MPS distribution. \$SUPDIR must be replaced with the absolute path to the μ MPS support directory. Make sure you know where it is installed in your local environment and alter this compiler directive appropriately.

3.9.7 Sample Module Decomposition

There isn't just one way to implement the functionality of this level.

One simple approach is to create:

- a module which takes care of starting the system (`boot.c`)
- a module for implementing the SSI (`ssi.c`)
- a module for the scheduler (`scheduler.c`)
- a module for exception management (`exception.c`), which could be replaced by two or more specialized modules (`syscall.c` `interrupt.c` `tlbtrap.c` `prgtrap.c`) in various combinations

3.10 Testing

There is a provided test file, `p2test.c` that will “exercise” your code.

The procedure for building it is the same described in § 2.5.

Remember that both Phase 1 and Phase 2 modules should be compiled and linked together `p2test.c` to get the full kernel.

Further Development

AMIKaya OS Project source code is not complete, in fact only Phase1 and Phase2 have been implemented. The author strongly encourages the development of feature improvements to provide all remaining phases to make AMIKaya truly complete.

Credits

The author would like to thank: Renzo Davoli, Michael Goldweber and Mauro Morsiani for the great work made with μ MPS, Kaya and AMIKE [1,2].

References

- [1] Renzo Davoli, Michael Goldweber, Mauro Morsiani, The Kaya OS Project and the MPS Hardware Emulator, 2005.
- [2] Mauro Morsiani, AMIKE OS specifications.
- [3] E. Dijkstra, The structure of the THE multiprogramming system, Commun ACM, 11(3), may 1968.
- [4] O. Babaoglu, F. Schneider, The HOCA operating system specifications, 1990.
- [5] O. Babaoglu, M. Bussan, R. Drummond, F. Schneider, Documentation for the CHIP computer system, 1988.
- [6] Renzo Davoli, Mauro Morsiani, Learning Operating System Structure and Implementation through the MPS Computer Simulator, 1999.
- [7] Renzo Davoli, VDE: Virtual Distributed Ethernet, In The Proceedings of Tridentcom, 2005.