Guide to the MIKONOS Operating System Project

Version 0.2

Renzo Davoli, Claudio Sacerdoti Coen

(based on the “Guide to the AMIKaya Operating System Project” by Enrico Cataldi)

January 15, 2008

1. Introduction

The MIKONOS Operating System (OS) described below is inspired to the previous experiences of Kaya OS[1], AMIKE OS[2] and AMIKaya OS[8] projects. All 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 and AMIKaya specifications introduced microkernel operating systems, based on the message passing facility. MIKONOS will be yet another microkernel operating system, based this time on an unusual variant of message passing inspired by Toth's primitives and the Qnix OS.

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

- Level 0: µMPS 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 1A – 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 ThreadBLK structures
- Level 3: the Nucleus (Phases 1B and 2 – described in Chapters 3 and 4). This level implements the thread scheduling, interrupt handling, message passing, deadlock detection, System Service Interface and its services. The boostrap phase (phase 1B) will be implemented and tested before the rest of the nucleus.

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 1A – Level 2: The Queue Managers

Level 2 of MIKONOS 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 what represent them at this level are thread control blocks (ThreadBLKs).

Each thread control block is identified by a thread identifier (TID) chosen from a set of 255 identifiers. An additional data structure needs to be implemented to map TIDs to pointers to ThreadBLKs.

```
typedef unsigned int status_t;      /* thread status */
typedef unsigned char tid_t;        /* thread identifier */
typedef struct tcb_t {              /* thread control block */
        tid_t           tid;        /* thread identifier */
        struct status_t     status; /* thread's status */
        struct tcb_t    *t_next,    /* pointer to next entry in the thread queue */
                        *inbox;     /* threads waiting to send a message to this thread */
        /* 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
- Resolution of TIDs to pointers to ThreadBLK elements
- Maintenance of ThreadBLK queues

2.1 ThreadBLK Allocation/De-allocation

One may assume that MIKONOS supports no more than MAXTHREADS concurrent threads; this parameters should be set to 32 (const.h). Thus, this level needs a number of MAXTHREADS ThreadBLKs to allocate from and de-allocate to. Assuming that there is a set of MAXTHREADS ThreadBLKs, one word in memory can act as a bitmap to identify the unused blocks. Entity allocation and de-allocation will be provided by these procedures, that can be directly accessed only from the functions described in Sect. 2.2.

```
void initTcbs(void);
```

Initializes the bitmap. This method will be called only from initTidTable.

```
void freeTcb(tcb_t *t);
```

Un-marks the element pointed by t in the bitmap. This function will be called only from killTcb.

```
tcb_t * allocTcb(void);
```

Returns NULL if all blocks are marked as used in the bitmap. Otherwise it marks an unmarked block and returns a pointer to it. All fields of the returned ThreadBLK (including the tid field) must be reset to default values (I.e. NULL and/or 0). ThreadBLKs get reused, so it is important that no previous value remains in a ThreadBLK when it gets reallocated. This function will be called only from newTcb.

Since MIKONOS 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 one array for the MAXTHREADS ThreadBLKs.

2.2 Resolution of TIDs to pointers to ThreadBLK elements.

Since MIKONOS is based on message passing, it is important to avoid immediate re-assignment of the TID of a terminated thread to a new thread. Otherwise, it is possible that a message will be erroneously delivered to a thread that was randomly assigned the TID of the deceased recipient. TIDs will be 8-bit integer values. 255 is not a valid TID. The next TID to be assigned is the successor (modulo 254) of the previously assigned TID, unless the TID is still in use. Several data structures (e.g. associative arrays, hashtables) can be used to implement the map from TIDs to ThreadBLK pointers.

```
void initTidTable(void);
```

Initializes the thread control block bitmap (by calling initTcbs) and the data structure used to map TIDs to ThreadBLK pointers. This method will be called only once.

```
tid_t newTcb(void);
```

Allocates a new ThreadBLK by calling allocTcb. If allocTcb fails, newTcb returns 255. Otherwise it returns the TID of the thread whose ThreadBLK has just been allocated. The rule to choose the TID is the one described above. The tid field of the ThreadBLK is set by the function to the chosen value.

```
void killTcb(tid_t tid);
```

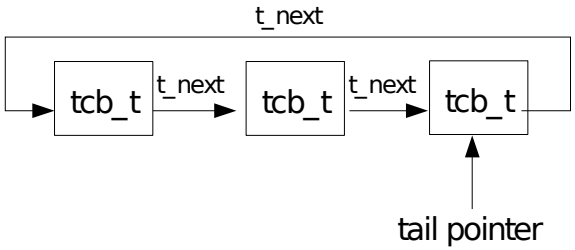
De-allocates (by calling freeTcb) the ThreadBLK identified by tid. The TID tid can now be reused.

```
tcb_t* resolveTid(tid_t  tid);
```

Returns the pointer to the ThreadBLK identified by tid.

2.3 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 insertBackThreadQ(tcb_t **tp, tcb_t *t_ptr);
```

Insert the ThreadBLK pointed to by t_ptr at the back of 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

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

Insert the ThreadBLK pointed to by t_ptr at the front of 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 * removeThreadQ(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 * outThreadQ(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 * headThreadQ(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

2.5 Nuts and Bolts

There isn't just one way to implement the functionality of this level. Regarding optimization, efficiency may be improved by introducing doubly-linked queues, adding eg. t_prev to ThreadBLKs. 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 three data structure, an array for ThreadBLKs, a one word bitmap for ThreadBLK allocation and an array or an hashtable to map TIDs to ThreadBLK pointers. Eg.:

HIDDEN tcb_t tcbTable[MAXTHREADS];

2.5 Testing

There is a provided test file, p1test.c that will "exercise" your code. You CANNOT modify p1test.c. 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 tid.c
mipsel-linux-gcc -ansi -pedantic -Wall -c p1test.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 tid.o p1test.o -o kernel
```

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

If one is working on a big-endian machine one should modify the above commands appropriately; substitute mips- for mipsel- and elf32btsmip.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 µMPS. 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 µMPS. 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 µMPS 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, µMPS 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 1B – Level 3: Bootstrap

The bootstrap code is still part of the Nucleus of MIKONOS. It is the code that is called by the ROM code responsible for initialization of the machine. Its role consists in:

1. Filling up the tables used by the ROM-Excpt handler (in Level 1) to handle exceptions. In particular, the bootstrap phase must provide the exception handler for: Program Traps (PgmTrap), SYSCALL/Breakpoint (SYS/Bp), TLB Management (TLB), and Interrupts (Ints). More details on the initialization of exception handling is given in Sect. 3.1.

   The final code for the exception handlers will be developed in Phase 2. To test the bootstrap phase during Phase 1B, the exception handler code will be provided in p1test.c.

2. Initializing the data structures used by the scheduler and the other parts of the Nucleus. In particular, the bootstrap code must invoke initTidTable(). During Phase 2, this code will be extended to fill the ready queue of the scheduler with the threads initially alive at the end of the bootstrap phase.

3. Calling the scheduler (I.e. invoking schedule()). To test Phase 1B, a fake schedule() function is provided by test1.c. The fake function simply exercises the bootstrap code by generating exceptions and veryfing that the right exception handlers are called.

Once the scheduler has been invoked, the role of the bootstrap code is finished. In particular, the schedule() function will never return. From now on, the only way to re-enter the Nucleus is through an exception handler.

3.1 Initialization of the New Areas in the ROM Reserved Frame

Every program needs an entry point (i.e. main()), even MIKONOS. The entry point for MIKONOS performs the nucleus initialization, which includes the population of the four New Areas in the ROM Reserved Frame. For each New processor state, you need to:

1. Set the PC to the address of your nucleus function that is to handle exceptions of that type
2. Set the $SP to RAMTOP. Each exception handler will use the last frame of RAM for its stack.
3. Set the Status register to mask all interrupts, turn virtual memory off, and be in kernelmode.

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.

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. Babaouglu, F. Schneider, The HOCA operating system specifications, 1990.
5. O. Babaouglu, 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.
8. Enrico Cataldi, Guidi to the AMIKaya Operating System Project, February 2007.