MiKABoO: implementation guide

Riccardo Maffei Teresa Signati Federico Bertani
Oleksandr Poddubnyy

June 24, 2017

Contents

1 Introduction

MiKABoO is a small project for the Operating Systems course at University of Bologna, in few words it is a small microkernel based on asynchronous message passing for educational purpose.
The aim of this paper isn't to present MiKABoO specifications or its general behavior, but to explain implementation choices.
The full project is divided in several "phases":

- Phase 1: data structures and low-level functions.
- Phase 2: thread scheduling, interrupt handling, message passing, deadlock detection, System Service Interface and its services.
- Phase 3: extension 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 phase adds user-level synchronization, message passing and a thread sleep/delay facility
- Phase 4: Networking.
- Phase 5: File System.
- Phase 6: Interactive shell.

Our implementation includes only the first two phases.
Each module is described in a section where a brief description is given along with major implementation choices if any.

Main modules:

- init
- Scheduler
- Interrupt
- Exception
  - TLB
  - Syscalls and breakpoints
  - PGM
- SSI

- Utilities
- Phase 1 data structure and functions
  - PCBs
  - TCBs
  - msgs

2 init

This module implements the first bootstrap stage including:

- Core data structures and scheduler initialization
- Population of the four New Areas in the ROM Reserved Frame
- Instantiation of thread for the SSI and insertion of its TCB in the Ready Queue
- Instantiation of thread for test and insertion of its TCB in the Ready Queue
- Update threads counter
- Calling the scheduler

2.1 ROM Reserved Frame population

For each exception type, a new state_t is created with the following settings:

- Set the PC to the address of nucleus function that should 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 the interrupts, turn virtual memory off, and be in kernel mode.

3 Scheduler

The scheduler module handles threads execution and timing related behavior. Scheduler implements the following structures:

- ***runningThread***: the thread currently running on the CPU
- ***readyQueue***: the queue of all threads ready to be executed
- ***waitQueue***: the queue of all soft-blocked threads
- ***pseudoClockList***: list of threads waiting to be waken up at next pseudo-clock tick
- ***totalThread***: the number of threads in the system
- ***softBlockedThread***: the number of threads waiting for I/O, trap passup handling, or end of an SSI service
- ***lastLoadTime***: time of day low in microseconds when last thread has been loaded
- ***timeSliceLeft***: time slice left to the running thread
- ***lastTickTime***: time of day low in microseconds when the last tick occurred
- ***totalTicks***: total number of pseudo-clock ticks since system boot

The scheduling algorithm is implemented using the Round Robin policy with simple deadlock detection.

3.1 Scheduling flow

1. convert and get current time of day in microseconds.
   This will be used everywhere to avoid time-related inconsistency.
2. handle pseudo-clock (as described in section 3.4)
3. get thread to execute
   (a) if there's a running thread then select it for scheduling
   (b) otherwise
      i. if there's a ready thread dequeue one
         A. reset time slice
      ii. otherwise
         A. if *totalThread* = 1 and the SSI is the only thread in the system: perform normal system shutdown, calling the HALT ROM routine
         B. if *totalThread* higher than zero but *softBlockedThread* is zero (a deadlock is detected): call the PANIC ROM routine
         C. if *totalThread* and *softBlockedThread* are higher than zero, some threads are waiting for I/O to complete: the system enter in wait state
            - set interval timer in order to handle pseudo-ticks
            - enable all interrupts
            - enter in CPU wait mode
4. set interval timer as the minimum between next tick or time slice end
5. set Last Load Time
6. load thread

3.2 *sched_init()*

Initializes all scheduler data structures.

3.3 *handle_accounting()*

Computes last elapsed time and charges to the parent process of the thread. Decreases the time slice and removes the thread from the *runningThread* if necessary

3.4 *handle_pseudoclock()*

Checks whether a pseudoclock tick has happened or not and updates necessary data structures. A clever math is used in order to avoid drift.
If a tick has happened send a fake message to all threads waiting for a tick in order to wake them up (send the number of total ticks as payload in order to emulate an SSI service reply).
A list of waiting threads is kept in order to reduce complexity.
Please note that when a thread is removed from the list, its hook is reset.

4 Interrupt

The interrupt module implements interrupt-related behavior and data structures.
Interrupt implements the following structures:

- ***threadsWaitingDevices***: Matrix to keep record of threads waiting for a device I/O.

4.1 *int_handler()*

This is the main interrupt handler routine called by the ROM when an interrupt is raised.
If a thread was running: saves the old state, decreases program counter and handles accounting (as described in section 3.3)
Detects the interrupt type (check is done in descending priority order to handle the interrupt with highest priority first):

- the first 2 interrupt lines are ignored according to specifications.
- the timer line is already handled by the scheduler
- all other lines are device related: call the right device handler

Calls the scheduler

4.2 *standard_device_handler()*

This function handles interrupts raised by devices.
When an interrupt is raised by a standard device, its status register is sent as message payload to the thread waiting for I/O (checking the matrix) emulating an SSI service reply.
The given line may have more than one interrupt raised (up to 8): they are handled in descending priority order.
Please note that terminals have 2 sub-devices: a reader and a writer. The latter has higher priority.

4.3 *get_highest_priority_interrupt()*

Given the interrupt line bitmap, this utility function returns the position of the first least significant bit that equals 1.

4.4 *init_interrupt_handler()*

Initializes the data structures used for interrupt handling, in particular it resets the matrix of threads waiting for I/O.

5 Exception

This module handles all system exceptions including:

- System calls and breakpoints
- Program traps
- Translation Lookaside Buffer traps

These are implemented as three different routines called by the ROM loading the CPU state from the ROM reserved frame.

5.1 *tlb_handler()*

This function handles TLB traps. In particular if the running thread has a TLB manager the trap is passed up (see section 5.4) and the accounting is handled (as described in section 3.3), otherwise the whole parent process is terminated.

5.2 *sys_bk_handler()*

This function handles system calls and breakpoints.
The "send" and "receive" system calls are implemented by the kernel while unrecognized syscalls are passed-up to syscall managers or may end in process termination.
Here some security checks are done.

5.2.1 Send implementation choices

Send is usually implemented calling the do send function (see section 5.5).
In this place we detect and catch messages coming from one of the managers of the recipient: in that case the message is supposed to be a TRAP_CONTINUE message, thus the recipient is woken up (no message is actually sent).

5.2.2 Recv implementation choices

A receive is considered successful if a message matching the given request is found in the message queue. If successful, the thread is scheduled again unless its time slice is expired.
Otherwise, if a message for the current thread is not found, the process is moved to wait queue but its program counter is not increased in order to make it call the syscall again once it wakes up. The thread will be automatically

woken up when a message, which satisfies the original syscall request, is sent to the thread.
This implementation also checks if the expected sender is dead (see sections 8.2 and 6.5 for more details) and sets errno accordingly.
Breakpoints are not implemented in this project.

5.3 *pgm_trap_handler()*

This function handles program traps in a similar way to the TLB handler (see section 5.1). In particular if the running thread has a program trap manager the trap is passed up (see section 5.4) and the accounting is handled (as described in section 3.3), otherwise the whole parent process is terminated.

5.4 *trap_passup()*

This function passes up the exception to the given manager. This function also moves the running thread to the wait queue and wakes the manager up. In particular it sends a pointer to the saved state of the offending thread as payload of the message.
This way the manager will have direct access to the state of the thread and can handle the trap by itself.

5.5 *do_send()*

This function enqueues the given message to the recipient and wakes him up if necessary. This is the lowest-level implementation of the asynchronous IPC send.
NOTE: this is the send used by the kernel, not the "send" system call. That system call is internally implemented upon this function.

6 System Service Interface

This module is implemented as a kernel-mode thread with an RPC-like structure.
Each time a message is received, a critical section is acquired disabling interrupts in order to process the request safely.
The request type is guessed looking at first 4 bytes of the message.
In order to parse request parameters correctly, we dynamically cast the given pointer to the guessed request type.

This module implements the following SSI services:

- GET_ERRNO
- CREATE_PROCESS
- CREATE_THREAD
- TERMINATE_PROCESS
- TERMINATE_THREAD
- SETPGMMGR
- SETTLBMGR
- SETSYSMGR
- GET_CPUTIME
- WAIT_FOR_CLOCK
- DO_IO
- GET_PROCESSID
- GET_MYTHREADID
- GET_PARENTPROCID

6.1 Get error number

This service returns the current value of the thread error number.
Note: each service will set errno according to its exit status.

6.2 Create process

This service creates a process child of the given parent and allocates its first thread with the given state.

6.3 Create thread

This service creates a thread with the given state and the given process as parent.

6.4 Terminate process

This service will terminate the given process and its whole process tree.
The termination of process tree is done in a DFS style.
The actual deletion will delete each thread and eventually the process itself.

6.5 Terminate thread

This service terminates the given thread, discarding all pending messages.
The thread is removed from the *threadsWaitingDevices* matrix if necessary (see sections 6.11 and 4).
The thread is removed from the pseudoclock waiting list if necessary (see sections 6.10 and 3.4).
Pending incoming messages are removed.
Pending outgoing messages are removed. They are kept in a list in order to reduce complexity (see section 8.3 and 8.2).
Threads waiting for a message from the terminating thread are automatically woken up and their expected sender is set as a special value (t_dead). They will perform a recv again and notice that the expected sender is dead (see section 5.2.2).
The thread is removed from all scheduler lists (if necessary) and then the tcb is freed (see section 8.2)

NOTE: if this is the last thread, this service invokes Terminate Process (as described in section 6.4)

6.6 Set program trap manager

This service sets the given PGM manager to the process of the given thread.
If the manager is already set, it terminates the process.

6.7 Set translation lookaside buffer trap manager

This service sets the given TLB manager to the process of the given thread.
If the manager is already set, it terminates the process.

6.8 Set system call manager

This service sets the given system call/breakpoint manager to the process of the given thread.
If the manager is already set, it terminates the process.

6.9 Get CPU time

This service returns the CPU execution time of the given process. This may include some kernel execution time if directly caused by the thread.

6.10 Wait for clock

This service suspends the given thread until the next pseudo clock tick.
The thread is inserted in a list in order to reduce complexity.

6.11 Do I/O

This service sets I/O parameters into device memory mapped areas.
A matrix is used for keep record of threads requesting I/O. This allows the interrupt handler to know which process to reply to.

In order to parse I/O parameters we dynamically cast the given pointer again according to the guessed device type.

In order to avoid race conditions main I/O steps are done in the following order:

1. Set I/O parameters.
2. Save thread ID in the matrix (see section 4)
3. Set I/O command

6.11.1 *getDeviceLineNumber()*

This function is used to get the device number and interrupt line number from a given device memory register address.

6.12 Get process ID

This service returns the caller thread's process ID.

6.13 Get my thread ID

This service returns the caller thread's ID.

6.14 Get parent's process ID

This service returns the given process' parent process' ID.

7 Utilities

This module implements some useful functions and macros.

7.1 *memcopy()*

This is a simple memcopy implementation without arch-dependent optimization.

7.2 TODLO_US

This macro returns the TODLO in microseconds. This is implemented using the actual conversion constant at runtime.

7.3 SET_IT_US

This macro sets the interval timer to the given value in microseconds. This is implemented using the actual conversion constant at runtime.

8 Phase 1 data structures and functions

This module implements main kernel data structures, some handling functions and some macros.

8.1 Process control block

This module defines the PCB data structure and some handling functions.
Processes are stored in a fixed-length array. In order to allow dynamic allocation we use a free list.

8.1.1 Process structure

PCB includes the following structures:

- ***p_parent***: pointer to the parent process (NULL if the root of the process tree)
- ***p_threads***: entry point of the threads list
- ***p_children***: entry point of the children processes list
- ***p_siblings***: link to siblings
- ***pgmMgr***: pointer to the process-specific PGM trap manager
- ***tlbMgr***: pointer to the process-specific TLB trap manager
- ***sysMgr***: pointer to the process-specific SYS/BK trap manager
- ***CPU_time***: total cpu execution time for the process, including all of its threads

8.1.2 *proc_init()*

This function initializes processes data structures and allocate the root process.
This is done inserting all the PCBs in the free list, and then extracting and allocating the root process.
Note: the root process is the only process with NULL as parent process.

8.1.3 *proc_alloc()*

This function extracts from the free list and allocates a new empty PCB as a child of the given parent process.
The new process has no children, threads, or managers and the CPU time is set to zero.

8.1.4 *proc_delete()*

This function deletes a process (properly updating the process tree links).
The process is deleted only if it has no children nor threads.
The PCB is then freed inserting it in the tail of the free list in order to reduce reincarnation odds.

8.1.5 *proc_firstchild()*

This function returns a pointer to the first child of the given process.

8.1.6 *proc_firstthread()*

This function returns a pointer to the first thread of the given process.

8.1.7 *extractPCB()*

This utility function is used to extract PCBs from the free list without the hassle of use the *container_of* macro every time.

8.2 Thread control block

This module defines the TCB data structure and some handling functions.
Threads are stored in a fixed-length array. In order to allow dynamic allocation we use a free list.

An unused TCB called *t_dead* is created separately as a dead special value. (see sections 5.2.2, 8.2 and 6.5)

8.2.1 Thread structure

- ***t_pcb***: pointer to the process, which the thread belongs to.
- ***errno***: error number.
- ***t_s***: CPU state.

- ***t_status***: thread status.
- ***t_wait4sender***: expected message sender.
  if the thread is waiting for a message (t_status == T_STATUS_W4MSG), this field contains the expected sender. NULL means anybody. (see sections 5.2 and 6.5)
- ***t_next***: link to others threads in the same process.
- ***t_sched***: link to other threads in the scheduling list.
- ***t_pseudo***: link to other threads waiting for a pseudoclock tick.
  This is added to reduce complexity in lookup in pseudoclock handling. (see sections 3.4 and 6.10)
- ***t_msgq***: entry point of the pending messages list.
- ***t_sentmsg***: entry point of the list of sent messages not "read" yet.
  This is added to properly handle thread termination reducing complexity. (see section 6.5)

8.2.2 *thread_init()*

This function initializes threads data structures. This is done inserting all the TCBs in the free list and setting their status to NONE.

8.2.3 *thread_alloc()*

This function extracts from the free list and allocates a new empty TCB as thread of the given process.
The thread will have no pending or sent messages.
Note: t_pseudo is initialized as pointing to itself (improperly using *INIT_LIST_HEAD* macro) in order to have a way to know whether it is in any list or not with low complexity and perform a safe deletion. (see sections 3.4, 6.10 and 6.5)

8.2.4 *thread_free()*

This function deallocates a thread (unregistering it from the list of threads of its process). Fails if the message queue is not empty. (see section6.5)
The TCB is then freed inserting it in the tail of the free list in order to reduce reincarnation odds.

8.2.5 *thread_enqueue()*

This function enqueues the given thread in the given queue.

8.2.6 *thread_qhead()*

This function returns the head element of a scheduling queue.

8.2.7 *thread_dequeue()*

This function dequeues the first element of a thread queue.

8.2.8 *extractTCB()*

This utility function is used to extract TCBs from the free list without the hassle of use the *container_of* macro every time.

8.3 Messages and message queues

This module defines the messages data structure and some handling functions.
Messages are stored in a fixed-length array. In order to allow dynamic allocation we use a free list.

8.3.1 Message Structure

- ***m_sender***: the message sender.
- ***m_value***: the message payload.
- ***m_next***: link to the next pending message in queue.
- ***m_tnext***: link to the next sent message in *tcb_t* list. (see sections 8.2 and 6.5)

8.3.2 *msgq_init()*

This function initializes messages data structures. This is done inserting all the messages in the free list

8.3.3 *msgq_add()*

This function extracts and adds a message to the destination message queue with the given data.
The message is also inserted in the sender sent message list. (see sections 8.2.1 and 6.5)

8.3.4 *msgq_get()*

This function retrieves a message from the destination message queue matching the given request, saving its payload to the given area if possible.

8.3.5 *msg_free()*

This function frees the message.
The message is removed from both the sent and pending lists. (see sections 8.2.1, 6.5 and 8.3.3)
The message is then inserted in the tail of the free list.

8.3.6 *extractMSG()*

This utility function is used to extract messages from the free list without the hassle of use the *container_of* macro every time.

9 Authors

Riccardo Maffei	riccardo.maffei@studio.unibo.it
Teresa Signati	teresa.signati@studio.unibo.it
Federico Bertani	federico.bertani@studio.unibo.it
Oleksandr Poddubnyy	oleksandr.poddubnyy@studio.unibo.it

This package contains some test scripts from Profressor Renzo Davoli and a list.h subset from the linux kernel.

10 License

The software and this documentation is released under the GNU GPL v2.0 license. This project contains a subset of list.h from the linux kernel released under GPL v2.0. A copy of the license is provided in the LICENSE file.