Implementation of µITRON Embedded Operating System Specification on top of L4 Microkernel

Anton Burtsev, anton@e1os.org

Supervisor: Ihor Kuz, ikuz@cse.unsw.edu.au

December 6, 2004

1. Introduction

The TRON project (TRON stands for "The Real-time Operating system Nucleus") was started in 1984 in the University of Tokyo. It was aimed to develop specifications for a pervasive computing environment. In order to take into account diverse requirements of a broad application domain, several different specifications were developed:

- ITRON (Industrial TRON) – specification for a real-time embedded operating system;
- BTRON (Business TRON) – specification for a desktop operating system;
- CTRON (Central and Communication TRON) – specification for communication and information exchange and processing;
- HMI (Human-Machine Interface) – specification standardizing ways of human-machine interaction.

The µITRON specification [6] was started as a sub-project within ITRON program targeted at the smallest systems with 8 or 16-bit microcontrollers. It was different from the base ITRON with a limited kernel functionality. Later, following the rapid hardware development, µITRON was adopted to be applicable to 32 microcontrollers. Therefore, µITRON is a specification of a real-time kernel for embedded systems with minimal hardware requirements.

µITRON provides set of essential primitives required to build a general embedded system. It defines a basic means of execution, communication, synchronization, simple memory management, scheduling, exception and interrupt handling. In order to match limited resources of simplest hardware, µITRON specifies neither memory protection nor memory translation. Therefore, kernel and all user tasks (a basic units of execution in µITRON) run in a single unprotected address space. Device drivers, network communication protocols or file systems can be implemented as user tasks and aren't specified by µITRON.

Functionality of a µITRON kernel is exposed by kernel objects. Usually, kernel object encapsulates some kernel resource, which is accessible to user through the system calls. Semaphores, mailboxes or time event handlers provided by µITRON kernel are typical examples of kernel objects. Kernel object can be created statically at system compilation time (µITRON provides special configuration mechanism called static system calls, which is intended to create objects at system compilation), or dynamically at run-time. µITRON objects are identified by unique IDs. Each type of object has an associated set of system calls, providing all necessary object operations.

Semaphore system calls (Figure 1) illustrate a common set of calls associated with each kernel object. µITRON API always provides methods for creating and deleting of object (`cre_sem` and `del_sem`).

Specification strictly separates system calls, which can be executed from tasks and from non-task context (e.g. interrupt and exception handlers). Usually, only non-blocking methods signaling object availability (i.e. releasing resource) can be invoked from non-task context. It prevents unpredictable behavior caused by blocking in non-task context. Therefore, object has a pair of signaling calls: one for task context (`sig_sem`) and one for non-task (`isig_sem`). System calls, which can block execution and put task to a waiting state always have three forms: infinite wait (`wai_sem`), poll (`pol_sem`) and wait with timeout (`twai_sem`). According to µITRON specification these service calls cannot be invoked from non-task context.

```
ER cre_sem  ( ID semid, T_CSEM pk_csem ); // Create semaphore
ER del_sem  ( ID semid );                 // Delete semaphore
ER sig_sem  ( ID semid );                 // Signal semaphore
ER isig_sem ( ID semid );                 // Signal semaphore (non-task call)
ER wai_sem  ( ID semid );                 // Wait semaphore
ER pol_sem  ( ID semid );                 // Poll semaphore
ER twai_sem ( ID semid, TMO tmout );      // Wait semaphore with timeout
```

Figure 1: System calls for manipulating µITRON semaphore (ID – semaphore object identifier, T_CSEM – pointer to the structure describing semaphore creation parameters (e.g. maximal and initial semaphore resource counters), TMO – wait timeout interval in microseconds)

Although µITRON is targeted on the smallest devices, there are still great diversity of hardware platforms falling into this category. In order to provide greater compatibility between µITRON applications, specification explicitly defines several function profiles aimed to enforce strict standardization in case where compromise between complete kernel functionality and limited hardware resources should be reached.

Each profile defines a subset of µITRON functionality (usually, it's subset of system calls). Current µITRON specification defines two profiles: Automotive Control Profile and Standard Profile.

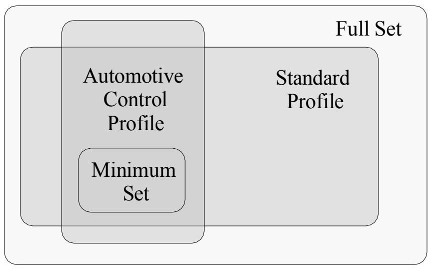

Figure 2: µITRON profiles

Standard profile is the most often implemented by µITRON kernels.

2. Implementation

This section describes implementation details of µITRON specification on top of L4 microkernel [7]. Presented implementation comply to the Standard Profile of µITRON 4.0 specification.

2.1. General architecture

µITRON kernel runs at user level in the L4 root-task address space (Figure 3). Kernel consists of a code implementing system calls and several kernel threads implementing interrupt and exception dispatching, user level scheduling and time event management. Since µITRON doesn't require memory protection, tasks run in the same address space as kernel. Task can invoke system calls just as ordinary function calls.

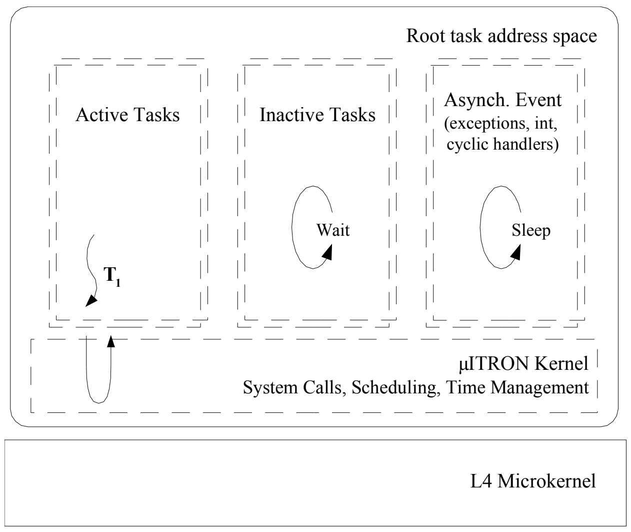

Figure 3: Generalized architecture of µITRON API on top of L4 microkernel

µITRON tasks are emulated with L4 threads executing task procedures cyclically. Inactive tasks wait for an activation IPCs. Asynchronous events (e.g. µITRON interrupt and exception handlers) are emulated generally in the same way as tasks, considering only that threads in a non-task context have higher priorities and thus can preempt running tasks.

2.2. Tasks

Execution flaw of a µITRON task is implemented by an L4 thread. Each task has a separate stack. Stack memory is either allocated by the kernel or passed from user level through the `cre_tsk` system call.

In order to control task preemption and optimize execution of several system calls, task uses user task control block (UTCB). It's a 35 bytes data structure containing information about nested critical sections, task state, activation counter and exception handlers. UTCB has to be accessible by both task thread and µITRON kernel. Therefore, natural place for UTCB is L4 thread's UTCB. However, on IA-32 architecture L4 UTCB doesn't provide sufficient space. Considering future protection extensions, it's undesirable to allocate separate page for storing UTCB. Therefore, µITRON UTCB is placed at the bottom of task stack. The address of the µITRON UTCB is stored in one of the user accessible L4 UTCB words.

Note that generally above described behavior is incorrect. µITRON task has predefined stack size. When stack memory is allocated on user level it cannot be enlarged by the kernel to reserve space for UTCB. Therefore, this can potentially lead to stack overflow.

In order to emulate µITRON task behavior, i.e. execute task procedure cyclically until task activation counter drops to zero, kernel uses a special wrapper code: kernel task procedure.

Upon creation of a new task µITRON kernel creates an L4 thread. Instead of passing execution to a task entry point, kernel sets up a stack with the UTCB and task creation parameters, and passes execution to a kernel task procedure trampoline code. The trampoline code is architecture dependent. Its goal is to dispatch parameters from the stack and pass them to a kernel task procedure. After that it passes execution to the kernel task procedure. Kernel task procedure checks activation counter stored in µITRON UTCB and if it's equal to zero (i.e. task is inactive) it performs L4 IPC receive system call with infinite timeout, otherwise it passes execution to the task entry point.

In order to activate a task, kernel tries to send the task an activation message with zero timeout. IPC failure means that task is executing task procedure. Therefore, kernel increments task activation counter expecting that task will check it on the next cycle.

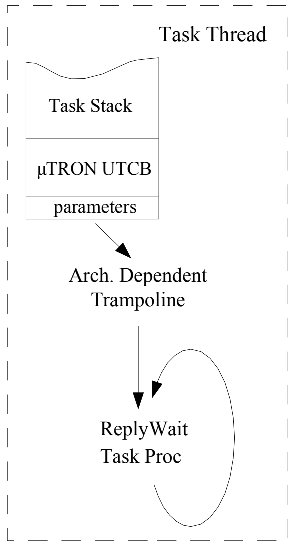

Figure 4: μITRON task.

2.3. Non-task context

Context is a μITRON abstraction consisting in restriction for a user to invoke system calls. Actually, it's aimed to allow optimization of kernel implementation. μITRON defines two types of context: task (for tasks execution) and non-task (for execution of interrupts, exception handlers and other asynchronous events). Restriction to invoke some system calls from non-task context allows, for example, processing of interrupts on the stack of currently executing task.

L4 provides a thread-based model for processing interrupts and CPU exceptions. In contrast, it has a very limited support for processing of asynchronous events on top of already existing threads. L4 provides ExchangeRegisters system call, which can suspend thread, save its state and transfer execution to an asynchronous event handler. However, there are several obstacles hindering utilization of this system call.

At first, ExchangeRegisters invocation is restricted to the address space of an affected thread. It cannot be invoked even from the privileged address space. Therefore, kernel has to create a special thread in task address space with the only goal to suspend the thread. This method works fine for large or medium address spaces, but it results in significant overhead in case of fine-grained address spaces, as they are in a μITRON implementation extended with protection mechanisms (each task will have a separate address space).

The second obstacle emerges from the complexity of thread state saving and restoring procedures, which, in fact, are equivalent to a thread switch mechanism already implemented in L4 kernel for different architectures. Therefore, it's natural to expect thread state management implementation from one of L4 user-level libraries.

μITRON Standard Profile defines execution in non-task context for interrupts, CPU and task exceptions, and time event handlers. Task exception is special μITRON communication mechanism. Task can define exception handler. Other tasks can raise exceptions and thus request asynchronous handler processing. Time event handlers are asynchronous events activated periodically or scheduled for a particular time.

Thread-based model perfectly matches needs of interrupts and CPU exception processing. However, task exceptions and time event handlers require implementation of a separate asynchronous processing model.

In current implementation non-task context is emulated by separate L4 threads, running at high priorities and thus capable to preempt tasks and provide asynchronous processing. Generally non-task context is implemented similarly to task context.

Non-task thread executes user defined event handler procedure in a ReplyWait cycle. Startup parameters are passed through the thread stack. They are dispatched by architecture dependent trampoline and then passed directly to a kernel handler procedure. Handler thread is activated either by timeouts or by incoming request messages. The only difference from task implementation is that non-task threads either don't have UTCB or share it with the associated task.

Interrupt and exception handler procedures provide also a compatibility layer required to implement L4 interrupt and exception protocols.

2.4. Synchonization

Synchronization is one of the most challenging and still open implementation problems of any operating system kernel. In real-time kernel this problem is even more complex because synchronization primitives have to provide not only efficient exclusive access to synchronized objects, but also a guaranty of wait-freedom and prevention of uncontrolled priority inversions. Moreover, synchronization primitives should provide reasonable trade-off between average and worst-case performance.

Several most widely used approaches to synchronization in real-time kernels include: priority inheritance, priority ceiling, delayed preemption and lock-free algorithms.

Priority inheritance provides good average performance in case of rare contentions, since only a minimal overhead is introduced by synchronization primitives. One of the most interesting implementations of a priority inheritance protocol was proposed in DROPS operating system [5]. It employs lock-free techniques to implement binary semaphore with priority inheritance. Unfortunately, priority inheritance can lead to poor worst-case performance in case of nested locks.

Immediate priority ceiling protocol solves the problem of worst-case performance of priority inheritance protocol. However, it introduces additional overhead caused by two priority changes.

Current μITRON implementation uses immediate priority ceiling protocol for task synchronization. In order to serialize priority change requests, a separate thread is used. μITRON UTCB stores nested critical section counter used to prevent unnecessary requests to serialization thread. To ensure correct execution of task wait and release, both operations are performed with raised priorities (Figures 5 and 6).

```
A1:  raise_prio();             // Send raise priority request
A2:  wq.enqueue(self);         // Enqueue task in a wait queue
A3:  wait();                   // Put task to wait
A4:  restore_prio();           // Send restore priority request
```

Figure 5: Wait operation pseudocode

```
B1:  raise_prio();             // Send raise priority request
B2:  wq.dequeue_first(&task);  // Dequeue first task from the wait queue
B3:  release(task);            // Release waiting task
B4:  restore_prio();           // Send restore priority request
```

Figure 6: Release operation pseudocode

Obviously, the major disadvantage of the above approach is poor average performance.

Fortunately, L4 provides delayed preemption as a reasonable alternative to priority ceiling protocol. Delayed preemption provides good both worst-case and average performance, since it avoids unnecessary context switches and delays high priority task only for a time required to complete critical section. It can also handle nested locks.

Delayed preemption protocol can be successfully applied for implementing synchronization in µITRON kernel in the way similar to the immediate priority ceiling protocol. In presented µITRON kernel, preference has been given to the immediate priority ceiling protocol only because of an incorrect implementation of delayed preemption in current version of L4 microkernel.

Major disadvantage of priority inheritance and priority ceiling protocols is that they affect whole system. In order to provide exclusive access, sensitive priority should be higher than priorities of all other threads in the system, which can potentially access protected data structure. In fact, this means that all threads including interrupts will be delayed for a time period required to complete a critical section, even if they don't actually need to access protected data structures. Potentially, this can result in impossibility to implement device drivers on top of µITRON kernel, since kernel will be unable to process interrupts in time. Therefore, precise answer to a question about applicability of priority ceiling and delayed preemption protocols to implementing synchronization primitives can be given only after performing careful benchmarking.

Above problems naturally result in seeking alternative ways to synchronize execution. One of the possible approaches lies in use of lock-free data structures and algorithms. In contrast to lock-based algorithms employing mutual exclusion, lock-free algorithms use atomic primitives to ensure consistency of a data-structure upon concurrent access. The algorithm is non-blocking (or lock-free) if any process can complete some action in a finite number of steps. Therefore, in a lock-free kernel any thread can at any time preempt lower priority thread, and successfully complete required operation in a finite number of steps, although at any time it can be preempted by other higher priority threads. This attractive property inspire application of lock-free algorithms for implementing kernel structures.

Unfortunately, development of lock-free algorithms is still in progress. It's hindered by the fact that any lock-free algorithm relies on the use of atomic primitives provided by a CPU. However, it's still unclear which atomic primitives ideally suit requirements of lock-free algorithms. For example, two successfully implemented lock-free kernels Synthesis [9] and Cache [1] are based on a very powerful primitive, which atomically updates two independent memory words (two word compare and swap). Unfortunately, this primitive is unavailable on most CPUs, that probably is the result of impossibility to implement it efficiently on most architectures. Therefore, most lock-free algorithms are developed on available yet less efficient primitives. Naturally, unavailability of a convenient primitive results in significant complexity on the upper (algorithmic) level. Thus, in order to solve ABA problem lock-free algorithms use complex reference counting or garbage collection techniques resulting in significant overhead even for relatively simple data structures like linked-lists. Furthermore, efficiency of lock-free algorithms given in many papers is deceiving. Many authors reporting good performance results of lock-free algorithms omit garbage collection in their benchmarks.

Over the course of the current work, a small investigation of applicability of lock-free algorithms to implementation of kernel data structures was conducted. As its result, it was planned to adopt hazard pointers approach [8] as a technique for safe memory reclamation occupied by a dynamically allocated lock-free objects. Providing safe memory management hazard pointers solve ABA problem and thus allow a uniform implementation of various types of lock-free data structures. Hazard pointers approach is capable to manage without significant performance degradation large number of objects. However, in order to be able to provide acceptable performance for a large number of threads base algorithm should be extended with a lock-free linked list for dynamic management of participating threads. In this work hazard pointers are extended with an algorithm given by Fomitchev et al. [3], which in turn is an improvement of algorithm given by Harris [4].

Obviously, the use of lock-free algorithms makes sense only in case they can provide kernel response better than their lock-based analogous. In case of a uniprocessor µITRON kernel this means that execution of operations on lock-free objects are faster than completion of critical sections.

Presented problems appeal for a powerful L4 user-level synchronization library capable to provide developers with a freedom to select synchronization primitives taking into account requirements of the target architecture (uni- or multi- processor) and desirable properties of synchronization algorithms (lock or wait freedom, average or worst case performance, etc.).

2.5. μITRON Objects

State of μITRON kernel is encapsulated by kernel objects. Each type of object has a set of associated system calls allowing applications to access their state. Objects are uniquely identified by an integer numbers.

μITRON objects are implemented as C structures. In order to provide identifier uniqueness and one-to-one correspondence between identifiers and object instances, each object is registered in two system structures:

- identifier registry;
- type object registry.

Standard profile doesn't require from μITRON kernel generation of unique identifiers. Instead, it specifies that kernel should only ensure identifier uniqueness. Therefore, system-wide identifier registry stores identifiers of all kernel objects preventing duplication of identifiers allocated by user-level applications.

The second structure (type object registry) provides a fast lookup among objects of a given type. Since object lookup is always performed within a system call defining implicitly a type of object, it's more efficient to lookup objects of different types in different registries than to perform a general lookup in one registry containing objects of all types.

Both registries are implemented as balanced Red-Black trees ordered by object identifiers.

2.6. Wait queues

Each μITRON object encapsulates certain resource. Most resources can be used simultaneously only by a finite number of tasks. All other tasks trying to access resource are transferred to a waiting state and placed to a waiting queue associated with the given object. μITRON specification defines two types of ordering in a waiting queue: FIFO and priority-based order.

In presented μITRON kernel FIFO queue is implemented by a singly-linked list. Priority-based queue is implemented as an array of singly-linked lists storing tasks with equal priorities and the hint pointing to the first (highest priority) list in the array.

Note, that the implementation of a priority-based queue is analogous to implementation of thread ready queue in Pistachio 3.x version of L4 microkernel. Obvious advantage of this implementation is simplicity. Unfortunately, it behaves poorly in case when queue contains sparse priorities.

Some μITRON objects require support of a fine-grained priority queues capable to manage efficiently large number of elements with sparse and dense priorities varying in intervals consisting of several thousands values. Moreover, priority queue should be stable and do not require preallocation of memory for the elements.

In order to satisfy these requirements a special data structure has been designed (Figure 7). Elements are simultaneously stored in a balanced red-black tree ordered by priorities and in a single-linked list. Balanced tree provides acceptable O(log n) addition of new elements while list ensures O(1) dispatching of the next highest priority element.

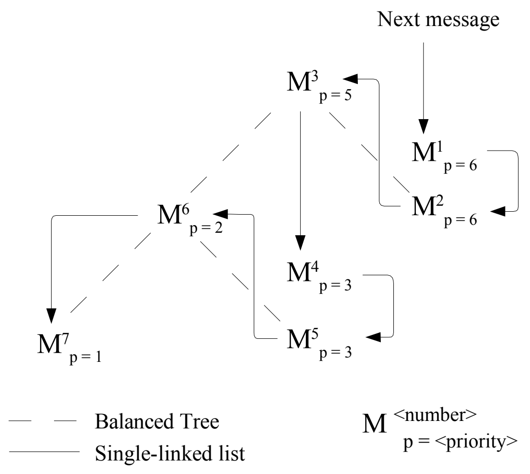

Figure 7: Fine-grained priority queue.

In order to implement stable algorithm, the tree stores only one entry for all equal priority elements. Addition of an element with an existing priority results in a substitution of a node in the tree by the added element.

2.7. Time event management

µITRON specification defines support for scheduling of time events. Time event is an execution of a user-defined handler at specified time. µITRON allows several forms of event scheduling: periodic execution with a given period and a single execution at specified time.

µITRON allows creation of arbitrary number of time event handlers. In order to manage them efficiently current implementation uses structure similar to the described above fine-grained priority queue. Time events are placed in a balanced Red-Black tree ordered by their execution times and in a single linked list. Balanced tree provides efficient O(log n) addition of new events while list ensures O (1) dispatching of a next event.

Special high-priority thread in a waiting cycle schedules and starts time event handlers. It can either create a separate thread for a long-running event handler, which execution can overlap next event, or can quickly execute short event handler and schedule next sleep. In both cases it supports accurate time management for event activations.

2.8. User-level Scheduling

µITRON specification defines preemptive, priority-based task scheduling. Its peculiarity is that task is never preempted due to a time slice exhaustion. Instead, task is executed until it is preempted by a task with a higher priority or until it becomes unable to proceed and is transferred to a waiting state.

L4 microkernel provides basic means of thread scheduling. It allows operating system to control thread execution order by assigning threads priorities and time slices. Unfortunately, the following requirements of µITRON API cannot be satisfied directly by L4 scheduling mechanisms and need implementation of a user-level scheduler:

1. µITRON specification requires ability to retrieve identifier of a currently executing task from non-task context.
2. API defines ability to implement round-robin scheduling via cyclic rotation of tasks in a ready queue.

L4 doesn't provide means for implementing custom scheduling policies. However, as it was told above, operating system on top of L4 can manipulate priorities of threads implicitly defining their execution order. Presented µITRON kernel adopts this approach (Figure 8).

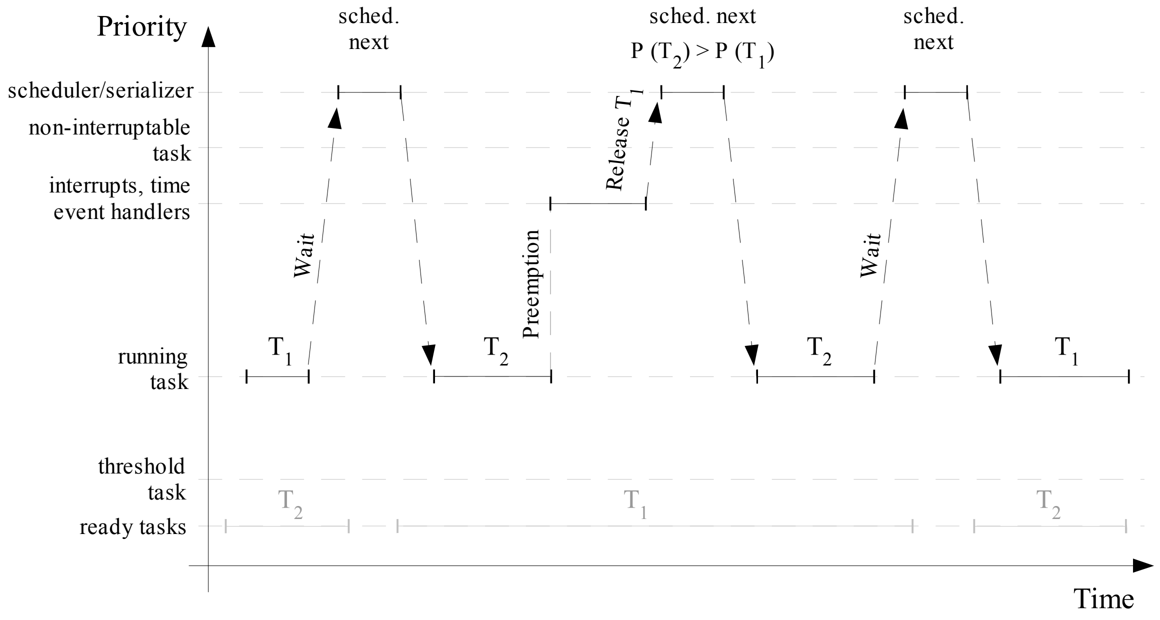

Figure 8. µITRON user-level scheduler.

In current implementation all threads have infinite L4 time slices, which ensures that tasks will be preempted only according to µITRON scheduling rules.

µITRON scheduler is implemented as a thread with the highest priority. Most of the time it sleeps waiting for requests from other threads. It's possible to trace µITRON task state transitions (e.g. between waiting and ready states) since they are all performed through invocation of µITRON system calls. System calls in turn are designed to report scheduler about state changes.

After creation task has the lowest priority. It cannot execute being blocked by the special system threshold task. Threshold task is used only to suspend not ready tasks. When task becomes runnable, scheduler raises its priority above priority of the threshold task. When task changes its state, scheduler lowers its priority back below the value of the threshold task thus suspending it effectively, and dispatches the next ready task.

Threads in non-task context run on higher priorities than tasks. Therefore, they are able to preempt running tasks. If running task wants to protect itself from preemption, according to immediate priority ceiling protocol, it asks scheduler to rise its priority above priorities of non-task threads.

µITRON scheduler maintains only one queue: priority-based queue of ready tasks. Non-ready tasks are stored in task object registry described in 2.5.

The obvious disadvantage of the proposed approach is it's inefficiency caused by large number of IPCs between tasks and the scheduler and intensive operations on the L4 thread ready queue. IPCs are required to report task state changes to the scheduler. L4 thread ready queue degrades to a list while storing large number of equal priority tasks.

In current µITRON implementation manipulations on task priorities partly serve as a workaround for a non-working thread suspend/resume method. Probably, optimized L4 ready queue and carefully implemented thread suspend/resume method will provide acceptable performance.

3. Real-Time Characteristics

Like any other real-time operating system, µITRON kernel should provide applications with limited and predictable response latencies. Generally this means to avoid uncontrolled priority inversions.

Current implementation uses immediate priority ceiling protocol for synchronization. This protocol avoids priority inversions and limits worst-case response latency time to the time required to complete a critical section. Completion of critical sections is also a source of unpredictability. Fortunately, all critical sections reside within the µITRON kernel. They are known and constructed to have execution times bounded by at least a logarithmic functions from a number of kernel objects.

At the same time, response latency consists not only from completion of a critical section but also includes time to make request to the user-level scheduler and perform thread switch. Moreover, it's most likely that this additional time is comparable or even larger than completion of the average critical section. Therefore, careful benchmarking is required to explore quantitative characteristics of presented architecture. Only this will give definitive answer about its viability as a real-time embedded system.

4. Protection Extensions

Protection Extensions is a separate µITRON specification extending µITRON 4.0 specification with means of memory protection and access control.

Following the base specification, protection extensions try to minimize requirements for a target hardware platform and reduce overhead introduced by memory management. Particularly, it requires no memory address translation.

Protection extensions introduce a notion of protection domain. It is a base unit of protection. µITRON kernel objects are created within protection domains. Kernel restricts access to protection domains by means of memory protection. Tasks as typical kernel objects are also created within protection domains.

Protection extensions introduce also a new type of object - memory object. Memory object presents a region of protected memory and can be used by tasks to securely share their data.

Access control vectors define operations, which task can perform on a particular kernel object.

Presented µITRON kernel doesn't implement protection extensions. However, it is designed to be easily extended to support them (Figure 9).

In order to implement protection extensions, µITRON kernel should be redesigned as follows: kernel and tasks are placed in different L4 address spaces. L4 spaces implement µITRON protection domains.

Tasks invoke system calls through the system call stubs sending send-and-receive IPCs to the kernel space. High priority kernel thread receives system call IPCs and dispatches them creating a new thread for each invocation. Newly created thread inherits priority of the invoking task.

Tasks and kernel can create and share memory objects by means of L4 memory mechanisms.

Note, that according to the above approach, invocation of µITRON system call results in a logical migration of the invoking thread to the kernel space. Migration here is a useful abstraction intended to ensure careful thread scheduling according to their priorities in both task and kernel spaces.

It's useful to distinguish thread migration as a logical abstraction and as an implementation paradigm. Thread migration as an abstraction can be successfully used to describe abstract thread of execution, which can cross boundaries of objects, address spaces or network nodes. At the same time, between boundaries it preserves its attributes (scheduling parameters, resource allocation policies, reflection policies, associated access rights etc.).

Abstract migration can be implemented on top of a message-based communication primitive, as proposed in current work, or can rely on thread migration mechanism. Thread migration is an approach to implementation of a cross-domain procedure call, according to which thread is capable to transfer its execution between address spaces upon cross-domain invocations.

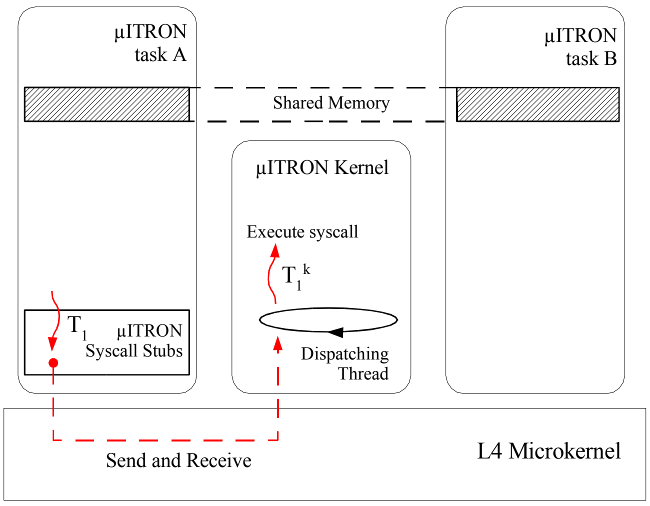

Figure 9. Protection extensions implementation on top of L4 microkernel.

Although, thread migration seems to be more natural for implementation of abstract migration than message-based approach, it's still unclear whether its efficient implementation is feasible in microkernel. The major problem of implementing thread migration in microkernel is allocation of stack for migrating thread. In message-based approach stack for a new thread in the invokee domain is allocated by the dispatching thread. Dispatching thread is a part of the operating system, it knows system memory model and therefore is capable to allocate stacks. Microkernel in contrast is unaware of operating system memory model and needs some upcall mechanism for stack allocation. However, it's unclear how to implement invocation of an untrusted upcall handler efficiently. The straightforward approach of implementing handler as a separate thread requires additional context switch for each allocation.

Above problems demonstrate a need for a more extensive research in the area of thread migration. Probably this research will lead to implementation of thread migration mechanisms in L4 microkernel.

5. Project Status

Over the course of this work a µITRON compliant kernel was implemented on top of L4 microkernel. Implementation fully conforms to the Standard Profile of µITRON 4.0 specification, except that it lacks µITRON system configuration files processing mechanism. As a consequence µITRON static system call invocation has not been implemented. Static system calls provide a method to create kernel objects statically defining them in system configuration files. In current implementation objects can be created only dynamically.

Presented µITRON kernel doesn't implement protection extensions. However, it is designed to be easily extended with protection mechanisms. Kernel subsystems do not rely on unprotected memory model. Kernel and tasks share only a carefully designed user task control block structure. Memory manager is ready to provide memory protection for µITRON protection domains and memory objects. System calls can be easily wrapped by L4 IPC stubs and provide cross-domain invocations.

Obviously, a sounding conclusion about applicability of presented real-time kernel to implementation of industrial embedded systems can be made only after conducting benchmark tests measuring various aspects of kernel behavior.

6. Conclusions and Future Work

Presented work is targeted at evaluation of L4 API completeness and its potential to be applied for development of embedded systems. In order to make this evaluation objective, popular µITRON specification representing real needs of industrial embedded systems has been implemented.

Generally providing a minimal set of flexible primitives for implementation of operating systems with different requirements, L4 has proved viability of its concepts. At the same time, some microkernel mechanisms discussed below turned to be still unrefined.

L4 scheduler is oriented mostly on development of time-sharing systems. L4 microkernel doesn't provide means for implementation of custom scheduling policies. This is an obvious inflexibility in L4 architecture. It's natural to expect from most systems scheduling requirements exceeding implementation freedom provided by priority and time slice parameters. Even simple scheduling rules of µITRON specification have resulted in use of a cumbersome and inefficient user-level scheduling workaround. Therefore, an extensive study targeted on development expressive scheduling primitives should be conducted.

L4 implicitly enforces system call access control (i.e. some system calls can be invoked only from the root-task address space, other can operate only within invoker address space, some use semantic dependencies between threads (e.g. one thread is defined to be the scheduler of another)). Obviously, this separation violates main principle of a good microkernel design - separation of policy and mechanism. It enforces policy. However, this policy is not sufficiently flexible to meet requirements of most operating systems. In protected µITRON this inflexibility has resulted in necessity to use additional threads aimed only to perform system calls within address spaces of target threads and send excessive redirection IPCs in order to comply to L4 invocation restrictions. Therefore, some flexible mechanism allowing operating systems to build their own access control policies is required.

It's natural to expect from L4 fully-fledged thread migration support. Migration seems to be more consistent representation of abstract execution than a chain of discrete object interactions. In real-time systems thread migration allows for example implementation of a fair scheduling between different address spaces [2].

L4 provides a very powerful memory management system allowing construction of complex virtual address space models. Unfortunately, it has a flaw allowing malicious task to violate real-time behavior of the system by creating a very long chain of recursive mappings. It's another argument to develop an access control mechanism restricting selectively memory operations.

It's obvious that unprotected µITRON cannot satisfy requirements of most contemporary embedded systems. At the same time Protection Extensions specification hasn't become a logical evolution of unprotected µITRON. Probably, its inflexibility and narrowness do not meet requirements of embedded software. The place of Protection Extensions as a successor of unprotected µITRON was occupied by T-Engine [10] and particularly by T-Kernel specifications.

T-Engine is a set of specifications for real-time embedded systems extending ITRON with notions of openness, portability, extensibility, and hardware-based PKI security. T-Engine envisages running Java and Linux (called T-Linux) as an application level extensions of T-Kernel. Naturally, T-Kernel can provide (although also as an extension) virtual memory management and protection.

Taking into account the amount of development efforts currently invested by former ITRON developers into T-Engine, it seems that implementation of T-Kernel specification is more promising alternative for L4 community than µITRON protection extensions.

7. References

[1] D. R. Cheriton and K. J. Duda. A Caching Model of Operating System Kernel Functionality. *In Proc. of the First Symp. on Operating Systems Design and Implementation*, pages 179--193. USENIX

Association, Nov. 1994.

[2] R. K. Clark, E. D. Jensen, and F. D. Reynolds. An architectural overview of the Alpha real-time distributed kernel. *In 1993 Winter USENIX Conf.*, pages 127--146, Apr. 1993.

[3] M. Fomitchev, E. Ruppert. Lock-free linked lists and skip lists, *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing,* St. John's, Newfoundland, Canada, 2004.

[4] T. Harris. A pragmatic implementation of non-blocking linked lists. *In Distributed Computing, 15th International Conference*, volume 2180 of Lecture Notes in Computer Science, pages 300--314. Springer-Verlag, October 2001.

[5] M. Hohmuth and H. Hartig. Pragmatic nonblocking synchronization for real-time systems. *In USENIX Annual Technical Conference*, Boston, MA, June 2001.

[6] ITRON Committee, TRON Association. MITRON4.0 Specification, ver. 4.00.00. URL http://www.itron.gr.jp/. 2002.

[7] The L4Ka Team. L4 experimental kernel reference manual, ver. X.2. URL http://www.l4ka.org. Dec 2002.

[8] M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.* 15(6): 491-504 (2004)

[9] C. Pu, H. Massalin, and J. Ioannidis, "The Synthesis Kernel," *Computing Systems 1*, 1, pp. 11-32 (Winter 1988).

[10] K. Sakamura, N. Koshizuka: T-Engine: The Open, Real-Time Embedded-Systems Platform. *IEEE Micro* 22(6): 48-57 (2002)