I. INTRODUCTION

Nowadays, most consumer electronics appliances have computing capability in order to retrieve data from sensors, to process the data, and to control devices based on the processed data. Such programs run on embedded operating system (OS) kernels, which provide basic execution primitives that can commonly be used by many appliances. The examples of embedded kernels include μITRON, pSoS, VxWorks, eCos, and so on. Those kernels typically do not provide protection domains because of their target platforms’ functional and resource limitations.

The recent emergence of digital appliances requires more advanced features, such as networking and GUI. Those features dramatically complicate the appliances’ software systems and increase their code sizes. Networked systems need to be prepared for attacks through the Internet. Since we cannot expect users to be system administrators of appliances, their software systems must be more robust than ordinary personal computer systems. Building such large, complex, and robust software systems on embedded kernels with the absence of protection domains is, however, very difficult since software bugs can cause system malfunction, data corruption, security breach, or even system destruction.

In this paper, we propose a system architecture that co-locates multiple embedded operating systems on a microkernel. The proposed architecture employs a microkernel to provide protected execution environments for the existing embedded kernels. In each protected execution environment, a kernel and its applications run just as they run on hardware since they share the same protection domain. Our approach can achieve the maximum reusability of the existing software resources including embedded OS kernels and their applications; thus, it protects the existing software resources. We have developed the system that consists of TL4 microkernel and a μITRON embedded kernel, and shows the feasibility of our approach through evaluations.

Abstract—Embedded operating system (OS) kernels provide basic execution primitives that can be used commonly by many appliances. The recent emergence of digital appliances requires more advanced features, such as networking and GUI. Those features dramatically complicate the appliances’ software systems and increase their code sizes. In order to develop such large and complex software systems, we propose a system architecture that co-locates multiple embedded operating systems on a microkernel. The proposed architecture employs a microkernel to provide protected execution environments for the existing embedded kernels. In each protected execution environment, a kernel and its applications run just as they run on hardware since they share the same protection domain. Our approach can achieve the maximum reusability of the existing software resources including embedded OS kernels and their applications; thus, it protects the existing software resources. We have developed the system that consists of TL4 microkernel and a μITRON embedded kernel, and shows the feasibility of our approach through evaluations.
Those features can protect the existing software resources, maintain the software quality, and save costs.

A. Related Work

In order to accommodate large and complex software systems, new embedded kernels that support protection domains have been created. For example, Linux is being enhanced to meet the requirements from embedded systems. The μITRON specification [11] was evolved to the μITRON4.0/PX Specification [12] that supports protection domains. A major drawback of using new kernels is that the existing applications and their supportive software require significant modifications because of compatibility issues. In other words, introducing protection domains requires a new kernel and new software. Since a significant amount of work has to be done for building up software resources, using a different kernel is apparently not a desirable solution. Our proposed architecture enables the reuse of the current kernel. By co-locating multiple kernels on a microkernel, we can enhance systems without sacrificing compatibility with the existing applications.

Our proposed architecture incorporates the hierarchical CPU scheduling to handle the multiple independent instances of a real-time kernel. A similarity can be found in the system that constructed a real-time kernel and a general purpose OS kernel on a multiplexer [1]. The system executes both kernels in the same protected domain; thus, it provides no protection between them. In contrast, our system executes the multiple instances of a real-time kernel in separate protection domains; thus, the kernel instances and their applications are protected in terms of data destruction and resource usage.

Other similar work includes user-level OS servers and virtual machines. User-level OS servers, such as UNIX server on Mach microkernel [6] and Linux server on L4 μ-kernel [7], run on a microkernel. The microkernel creates separate protected domains for the OS server and its user processes. The microkernel provides and schedules threads to execute them. Since thread scheduling is done by the microkernel, the scheduling is not hierarchical. Although our system executes an embedded kernel at the user level, it is not an OS server since the kernel and its applications share the same protection domain. The embedded kernel performs the scheduling of its applications; thus, the scheduling is hierarchical.

Virtual machines have a long history of researches [5] and the actual production uses by mainframe computers [2]. Recent PC processors became fast enough to enable virtual machines without hardware support found in mainframes [8], [10]. By taking advantage of running OS kernels as applications, virtual machines can be used as a tool to enhance security features [3], [4]. OS kernels running inside virtual machines are completely separated from their underlying software layer that executes those virtual machines; thus, the scheduling is hierarchical. There is, however, a significant difference between those virtual machines and our proposed architecture. Our targets are embedded systems while those virtual machines target servers. Since embedded systems require the timeliness of processing and responses, different scheduling policies and mechanisms are required.

B. Paper Organization

The rest of this paper is organized as follows. Section III describes the overview of the system and how it can be utilized. Section III describes the design of the system, and Section IV shows the current status of the implementation and shows the preliminary evaluation results. Finally, Section V summarizes the paper and presents the future work.

II. SYSTEM OVERVIEW AND USAGE SCENARIOS

Figure 1 depicts the overall architecture of the system. This system consists of TL4 microkernel, the multiple instances of a μITRON kernel. Multiple applications can run within a single instance of a μITRON kernel while they are not protected from each other since they share the same protection domain. There can be μITRON kernel instances for the provision of services. Their own protection domains can be dedicated to running servers for security and protection. Applications can access services provided by servers through server proxies. Server proxies hide communications between the protection domains of applications and servers.

Only TL4 microkernel executes in the privileged mode directly on top of hardware. TL4 microkernel provides protection domains, threads, and IPC. A protection domain and threads constitute an execution environment of a single μITRON kernel instance and their applications. Threads provided by TL4 microkernel execute in an execution environment in the user mode, so that only limited and controlled access is granted to the μITRON kernel instance in it. Since different protection domains are allocated for applications and servers, the misbehaviors of applications do not cause data destruction in servers’ protection domains.

Figure 2 shows an example system setup that can effectively utilize multiple protection domains. In the example setup, there are four protection domains for applications, network services, personal data file services, and local device services. It is important to dedicate a protection domain to personal data file services in order to isolate personal data files from any illegal access and only to allow conforming ways to retrieve such data. Network services are isolated in another protection
domain since a network subsystem is the most likely an entry point for a system to be compromised. By limiting and controlling access to personal data file services from network services, severe security breach can be avoided. Local device services take another protection domain since they implement the drivers of devices shared by applications and the other services.

There is actually no need for applications and servers to share protection domains. By sharing a protection domain, however, we can make a system consume less resources. The most important point for applications to have their protection domain is to protect personal data file services. Let us consider a case that an application misbehaved and killed the other applications. Such misbehavior should not happen, but there can be a corner case that were overlooked during quality assurance tests. As far as there was no data loss, applications can be simply lunched again hopefully without users’ awareness. If data was lost or destroyed, it is a problem for all users and the system needs to be updated to fix the problem.

When a user installs a third party application program (or a free application program on the Internet for an even worse case), it is desirable to dedicate a protection domain to it. There is no warranty that the installed application is not some kind of a virus or spyware. It is much safer to isolate such a program from the other application and server programs. Another similar example use of protection domains is for debugging. It is difficult to find bugs that are caused by overwriting data of other programs if several programs share the same protection domain. Testing a single program may not reveal such bugs since they do not affect the execution of the program itself. At the time of debugging, we can take advantage of a setup in which all programs use different protection domains. In such a setup, only necessary memory regions are allocated for each program; thus, out-of-range memory references can be easily detected.

III. DESIGN

This section describes the details of the enhancements and modifications made to TL4 microkernel and a μITRON kernel in order to run multiple μITRON kernel instances on TL4 microkernel. TL4 microkernel is based on L4 microkernel [9], and is enhanced to enable the execution of multiple μITRON kernel instances. TL4 microkernel inherits L4 microkernel’s simple abstractions, that include threads, protection domains, memory pages, and IPC. Note that in the rest of this paper we use threads to refer to TL4 microkernel’s execution entities and tasks or applications to refer to μITRON kernel’s execution entities.

A. μITRON Kernel on TL4 Microkernel

A μITRON kernel is a simple embedded real-time kernel that provides real-time tasks, synchronization and communication mechanisms, and device drivers. An implementation of the kernel can be divided into the machine independent and dependent parts. The machine independent part includes the common mechanisms and policies of the kernel while the machine dependent part includes platform dependent mechanisms and device drivers. In order to bring a μITRON kernel on TL4 microkernel, the machine dependent part needs to be modified. Since the maximum reusability of the existing software including the kernel is our major goal, the modifications need to be minimized. Therefore, we introduce a layer, called the processor emulator, that emulates the hardware and encapsulates the differences from the hardware.

Figure 3 depicts the structure and elements of a μITRON kernel on TL4 microkernel. A μITRON kernel on TL4 microkernel consists of three layers, the machine independent part, the dependent part, and the processor emulator. Threads provided by TL4 microkernel execute a μITRON kernel. Threads are used in two different ways. One is for the execution of the all three layers. We call this thread the main execution thread. The others are for handling interrupts and execute only device drivers in the dependent part and the processor emulator. We call those threads interrupt emulation threads. Interrupt emulation threads run at higher priority levels than the main execution threads in order to emulate interrupts. The processor emulator manages those two types of threads in order to emulate the hardware, and thus enables the execution of multiple μITRON kernel instances. The processor emulator deals with interrupts, time management, scheduling events, and the idle state. Their details are the following.

1) Controlling Interrupts: Disabling interrupts is a simple yet efficient way to protect critical sections for single CPU
systems. When a μITRON kernel instance runs on top of TL4 microkernel, it cannot disable interrupts of the CPU. If it does, the other instances will not receive interrupts, either. Since there can be more important and urgent tasks in the other instances, only interrupts to a certain μITRON kernel instance should be disabled. Therefore, interrupt disabling needs to be emulated by some means.

Our μITRON kernel on TL4 microkernel emulates interrupt disabling by introducing a flag in the processor emulator. If the flag is set, it indicates interrupts are disabled. When an interrupt occurs and its interrupt emulation thread starts its processing in the processor emulator, it checks if the flag is set or not. If it is set meaning interrupts are disabled, the interrupt emulation thread yields the execution and waits for a message notifying interrupts are enabled. In other words, interrupt requests are queued for later processing. This way does not require any interventions to invoke TL4 microkernel nor to program an interrupt controller. The processor emulator handles interrupt disabling by managing interrupt emulation threads. Since it just requires the processor emulator to set the flag, it is very lightweight. The interrupt disabling emulation in the processor emulator is invoked by calling a certain function in the processor emulator to set the interrupt disabling flag.

2) Time Management: A kernel manages its time usually relying on periodic interrupts from a timer device. When multiple μITRON kernel instances run on top of TL4 microkernel, we need to consider the scheduling of the timer interrupt emulation threads for those kernel instances. There are at least the main execution thread and the timer interrupt emulation thread for a single μITRON kernel instance. Those threads are runnable only when their μITRON kernel instance is scheduled to run by TL4 microkernel. If the timer interrupt emulation thread is executed every time it becomes runnable, the timer interrupt handler is executed to update the time of its kernel instance; thus, the kernel instance can keep its time updated. If the timer interrupt emulation thread is not executed by the time when the next timer interrupt should happen, the time of its kernel instance is not updated; thus, the kernel instance cannot keep its time up to date. This can happen if there are higher priority kernel instances that are scheduled before the kernel instance in question.

We can deal with the above problem by having the processor emulator emulate the timer interrupts that occurred while a μITRON kernel instance was blocked by controlling the number of times the timer interrupt handler is executed. When a timer interrupt occurs, before calling the timer interrupt handler of a μITRON kernel, the processor emulator examines if the handler has missed any of its past timer interrupts. If it happened, the processor emulator calculates how many interrupts the handler has missed the dues. Finally, the processor emulator calls the timer interrupt handler the necessary times to catch up with the current time. Note that the timer interrupt emulation thread runs at the highest priority level; thus, no other threads can run before it finishes its time management.

3) Dealing with External Scheduling Events: Scheduling needs to be done in two cases. One is an internal scheduling event that happens when the current execution voluntary relinquishes the CPU by calling the scheduler. Since internal events can be handled within a μITRON kernel, there is no difference even if it runs on TL4 microkernel. The other is an external scheduling event that happens when an interrupt occurs and a higher priority task wakes up. External events require interrupt emulation threads to control the execution of the main execution thread. An interrupt is processed by an interrupt emulation thread but tasks are executed on the main execution thread; thus, the current instruction pointer of the main execution thread needs to be changed from an interrupt emulation thread. TL4 microkernel provides a system primitive for that purpose. The primitive can change the instruction and stack pointers of the thread specified by the argument and retrieves the old values of the instruction and stack pointers. Those old values are saved for later resumption.

4) Dealing with Idle State: When all tasks are blocked and there is no task to run in a μITRON kernel, the kernel falls into the idle state. When a μITRON kernel on top of TL4 microkernel finds that it falls into the idle state, the main execution thread needs to block in order to avoid disturbing the other instances’ execution by just spinning. The main execution thread invokes TL4 microkernel to wait for a notification message of a scheduling event. When an interrupt occurs and it causes an scheduling event, an interrupt emulation thread makes the main execution thread call the scheduler as described in the previous section. At that time, the interrupt emulation thread examines if the main execution thread is in the idle state or not. If not, then no action is needed. If it is in the idle state, the interrupt emulation thread sends a notification message to it and wakes it up from the idle state.

B. TL4 Microkernel

This section describes the enhancements made to TL4 microkernel in order to execute multiple μITRON kernel instances on it.

1) Scheduler: TL4 microkernel’s execution entities are threads. TL4 microkernel’s scheduler first selects which μITRON kernel instance to run among multiple instances. After that, the scheduler needs to determine which thread to run. Since there are multiple threads that execute a μITRON kernel instance, each instance has a thread queue that maintains runnable threads of the instance. Threads are maintained in priority order. The priority of threads is only effective in each μITRON kernel instance, so that the thread priority levels of different μITRON kernel instances are never compared.

2) Scheduling of Interrupt Emulation Threads: An interrupt emulation thread waits for an IPC message from a certain interrupt source, and an interrupt wakes up the thread. There are the following three states of a μITRON kernel instance, and in each state an interrupt emulation thread needs to be treated differently in TL4 microkernel:

- The instance is running: In this case, either the main execution thread or the other interrupt emulation thread is running. The priority level of the awaken interrupt
emulation thread is compared with the current thread. If the awaken interrupt emulation thread has higher priority, it preempts the current thread. If it has lower priority, it is inserted into the thread queue.

- The instance is runnable but not running: In this case, the awaken interrupt emulation thread is simply inserted into the thread queue. Since the instance is runnable, there is no other thing to be done.
- The instance is not runnable: In this case, the instance is in the idle state. The awaken interrupt emulation thread is inserted into the thread queue, and the instance is marked runnable. When the instance is scheduled, the interrupt emulation thread runs. If a scheduling event happens, the main execution thread resumes its execution (see Dealing with Idle State in Section III-A.4).

IV. EVALUATION

We have just finished our first implementation of the system described in this paper. TL4 microkernel was implemented based on L4Ka::Hazelnut, which is a version of L4 µ-kernel. Our µITRON kernel on top of TL4 microkernel was implemented based on TOPPERS/JSP, which is an open source µITRON kernel compliant to the µITRON4.0 specification. The rest of this section shows the evaluation results obtained from the current implementation. We first compare the memory footprints for our µITRON kernel on TL4 microkernel and the original µITRON kernel to find out the memory overhead. Next, we show the measurement results of invocation latencies from a simple application setup. All measurements were performed on IBM ThinkPad X23 Laptop PC with Intel Mobile Pentium III 866MHz CPU. The measurements used the high-resolution timestamp counter built in the CPU. All times shown below are the average of costs that were measured 500 times.

A. Memory Footprints

Table I shows the memory sizes consumed to run a single instance of our µITRON kernel on TL4 microkernel. If multiple instances are created, the memory sizes for the µITRON kernel are multiplied by the number of its instances. The memory footprint of a µITRON kernel instance on TL4 microkernel is 63KB, which is slightly smaller than the footprint of the original µITRON kernel. TL4 microkernel, however, takes 47KB; thus, the total memory footprint of a µITRON kernel instance on TL4 microkernel is 45KB larger. Since protection domains are needed to construct large systems in order to deal with software complexity, such memory overhead is negligible. Although we cannot compare those memory footprints with the embedded versions of Linux kernel, they require significantly larger memory sizes.

B. Invocation Latencies

We measured the latencies from the software entry point of interrupt processing to the invocation of an interrupt handler and application tasks. Figure 4 (a) and (b) show the measurement setups for the original µITRON kernel on hardware and the µITRON kernel on TL4 microkernel, respectively. The arrowed lines in the figures indicate the flow of control starting from receiving an interrupt. There are two applications, Application Task 1 and 2. Application Task 1 receives a character from a serial line device, and Application Task 1 passes the received character to Task 2. T1, T2, and T3 indicate the invocation times of the serial device interrupt handler, Application Task 1 and 2 by receiving a character, respectively.

Table II shows the measurements results. The table shows T1, T2, and T3 described above for both cases of the µITRON kernel on TL4 microkernel and the original µITRON kernel on hardware. The results interestingly shows that the µITRON kernel on TL4 microkernel outperforms the original µITRON kernel for all three points although the differences become smaller as the execution goes forward.

The latency to invoke the interrupt handler in the µITRON kernel shows the most significant difference between the two cases. The case of the µITRON kernel on TL4 microkernel involves the extra costs of the context switching to the interrupt emulation thread and crossing the privilege/user mode boundary. The results show that the extra costs incurred to run the µITRON kernel on TL4 microkernel are negligible for the latency to invoke the interrupt handler.

The execution flow from T1 to T2 involves the interrupt handler’s cost to read an input character from the serial line device, to wake up Application Task 1, and to switch the context to it. The costs from T1 to T2 are 6.32 μsec for the µITRON kernel on TL4 microkernel and 6.08 μsec for the original µITRON kernel. The difference includes the costs of TL4’s system primitive to change the instruction and stack pointer of the main execution thread, and the context switching from the interrupt emulation thread to the main execution thread.

The execution flow from T2 to T3 involves only the context switching from Application Task 1 to Application Task 2 using the semaphore primitives provided by the µITRON kernel. There is no difference in the execution path between the two cases. Since the application task switching is performed on the main execution thread for the case of the µITRON kernel.
on TL4 microkernel. There is, however, the difference in the cost between them. The costs from T2 to T3 are 1.72 μsec for the μITRON kernel on TL4 microkernel and 0.89 μsec for the original μITRON kernel. The μITRON kernel on TL4 microkernel is approximately twice slower than the original μITRON kernel. More investigations need to be performed to find out the reasons for this difference.

V. SUMMARY

We presented an alternative approach to introduce protected domains to the existing embedded systems. Our approach employs a microkernel to provide protected execution environments for the existing embedded kernels. In each protected execution environment, a kernel and its applications run just as they run directly on hardware since they share the same protection domain. Our approach can achieve the maximum reusability of the existing software resources including embedded OS kernels and their applications. We have developed a prototype system that consists of TL4 microkernel and the μITRON embedded kernel, performed preliminary evaluations, and showed the system worked as well as the original μITRON kernel.

We are currently working on creating more realistic and practical setup, and more accurate and various performance evaluation results will be completed using them.

REFERENCES