Operating Systems for Building Robust Embedded Systems

Tatsuo Nakajima†, Midori Sugaya†, Shuichi Oikawa‡
†Department of Computer Science, Waseda University
‡Graduate School of Systems and Information Engineering, University of Tsukuba
{tatsuo, doly}@dcl.info.waseda.ac.jp, shui@cs.tsukuba.ac.jp

ABSTRACT

Embedded systems will become more and more complex in the near future. In Japan, μITRON and embedded Linux are popular for building embedded systems. Since μITRON does not support memory protection, the bugs in an application may cause serious system failure. Also, embedded Linux cannot avoid an application to monopolize the entire CPU capacity due to its bugs. Therefore, we need to increase the robustness of embedded systems at the operating system level for developing future complex embedded systems.

We have developed two operating systems to increase the robustness of embedded systems even if applications contain serious bugs. The first operating system supports multiple execution of μITRON. Each application is executed on a different μITRON kernel. Therefore, if an application is crashed, other applications that are in different address spaces have no effect. The second operating system enhances the resource management of embedded Linux. Our system monitors an application's CPU usage and stops the execution if the usage exceeds the specified capacity.

1. INTRODUCTION

In the future, ubiquitous computing environments[6] will change our lives dramatically. The vision of ubiquitous computing environments is to acquire information in our environments that are not available before by using sensor technologies[2]. Also, the environments will make it possible to control many everyday objects by embedding very small and cheap computers. One of the most important issues to realize ubiquitous computing is to integrate a real world and a cyber space in a seamless way. This makes it possible to merge bits and atoms. Thus, software infrastructure for ubiquitous computing should provide a world model that provides a model of our world, which can be accessed by a program, and an application can change its behavior and change the real world by accessing the model. Also, a model in a cyber space can be manipulated by a physical object.

Information appliances are important elements for realizing ubiquitous computing visions. Nowadays, most consumer electronics appliances have computing capability in order to retrieve data from sensors, to process the data, and to control devices. The recent emergence of information 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 though 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 is, however, very difficult since software bugs can cause system malfunction, data corruption, security breach, or even system destruction.

Information appliances will become more and more complex in the near future. In Japan, μITRON and embedded Linux are popular to be used in embedded systems. Since μITRON does not support memory protection, the bugs in an application may cause serious system failure. Also, embedded Linux cannot avoid an application to monopolize the entire CPU capacity due to its bugs. We need to increase the robustness of embedded systems at the operating system level for developing future complex embedded systems.

We have developed two operating systems to increase the robustness of embeddedsystems even if applications contain serious bugs. The first operating system supports multiple execution of μITRON. Each application is executed on a different μITRON kernel. Therefore, if an application is crashed, other applications that are in different address spaces have no effect. The second operating system enhances the resource management of embedded Linux. Our system monitors an application's CPU usage and stops the execution if the usage exceeds the specified capacity.

The remaining of the paper is structured as follows. Section 2 presents two operating systemμITRON and embedded Linux that are currently widely used in Japanese embedded systems. In Section 3, we present a microkernel-based operating system that executes multiple μITRON operating systems. Section 4 describes an accounting system on Linux. It protects a system from monopolizing the CPU capacity from malicious applications. In Section 5, we conclude the paper.

2. LINUX AND μITRON

In Japan, a lot of industrial embedded products have adopted the *ITRON specification operating system(μITRON)*. μITRON is not an actual operating system implementation. It specifies the kernel interface, and many vendors has been implemented the specification for their products. Also, many RTOS vendors sell products that implemented the specification. The specification contains basic functionalities such as scheduling, thread managements, and simple inter process communication. Thus,respective companies have been implemented a lot of software on the operat-

Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'05)
0-7695-2347-1/05 $20.00 © 2005 IEEE

ing systems. Currently, the operating systems have been adopted by many products such as digital televisions and cellular phones that are developed in Japan.

However, most of embedded systems have became very complex now. For example, a current cellular phone in Japan contains a Web browser, a Java virtual machine, e-mail software, and camera software. On μITRON, these software modules are running on a single address space. To implement these software modules in a robust way, we need a more powerful operating system. Linux currently supports various CPU architectures, and kernel modules can be loaded dynamically. The characteristics are very suitable for embedded systems. Therefore, many industries have considered that using Linux for their products provides big merits for them.

Japan Embedded Linux Consortium (www.emblix.org) has been developed the Linux on ITRON specification to combine the advantages of both μITRON and embedded Linux. Linux on ITRON executes Linux as one of threads of μITRON. Therefore, both μITRON applications and Linux applications run on a single system. The fast response time is ensured for μITRON applications and many Linux applications can be reused on Linux. However, software bugs in μITRON applications may crash the entire system. Also, malicious applications may monopolize the entire system capacity. We have developed our operating systems to solve the problems.

3. A MICROKERNEL-BASED OPERATING SYSTEM FOR BUILDING ROBUST EMBEDDED SYSTEMS

In this section, we propose a system architecture that colocates multiple embeddedoperating systems on a microkernel. The proposed architecture employs a microkernel to provide protected execution environments for the existing embedded kernels that have noprotection mechanism. In each protected execution environment, a kernel, its applications, and servers share the same protection domain and run just as they run directly on hardware; thus, there is no need to port the existing software to run on different operating systems. The microkernel supports multiple protected execution environments, so that we can concurrently run the multiple instances of an embedded kernel along with applications. Applications and servers can be decoupled to different protection domains, so that they can be isolated to reinforce reliability and security. In this case, applications can use server proxies to use services provided by servers with the same API. The microkernel performs the scheduling of embedded kernel instances. We provide two scheduling policies in the microkernel for different purposes. One of those policies can be selected at the time of system configuration.

In order to show the feasibility of our approach, we are developing a system that consists of *TL4* microkernel and a μITRON kernel. *TL4* microkernel is being developed based on L4 μ-kernel [5]. We chose to apply our approach to a μITRON kernel since embedded OS kernel implementations compliant to the μITRON specification [9] are the most popular in Japan. In this paper, we refer to an implementation of an embedded OS kernel following theμITRON specification as a μITRON kernel.

The features of this system are summarized as follows:

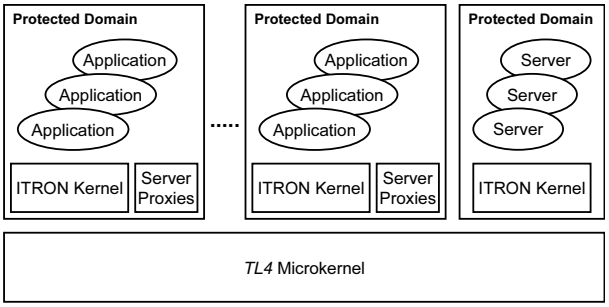

Figure 1: Overview of System Architecture

- This system enables the provision of protected domains without affecting the compatibility of the kernel APIs by employing a microkernel.
- It can achieve the maximum reusability of the existing software resources including embedded OS kernels and their applications.
- It enables the schedulability analysis of real-time tasks on an embedded OS kernel, so that it can guarantee that those tasks are scheduled in a timely manner.

Those features can protect the existing software resources, maintain the software quality, and save costs.

3.1 System Overview

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.

3.2 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 μ-kernel [5], and is enhanced to enable the execution of multiple μITRON kernel instances. *TL4* microkernel inherits L4 μ-kernel's simple abstractions, that include threads, protection domains, memory pages, and IPC. Note that in the rest of

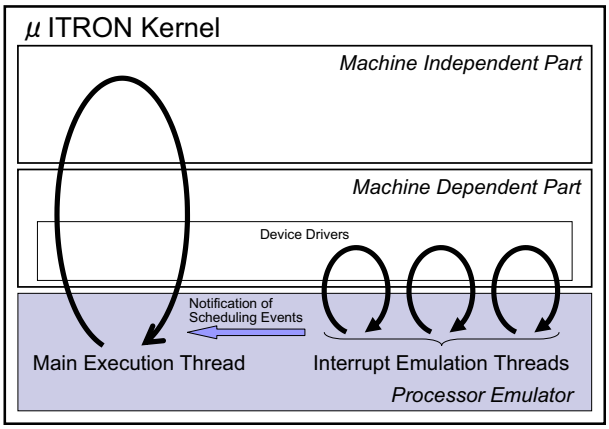

Figure 2: Structure and Elements of μITRON Kernel on *TL4* Microkernel

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.

3.2.1 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 2 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.

3.2.1.1 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.

3.2.1.2 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 that emulates the timer interrupts 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 dues. If it happened, the processor emulator calculates how many times the handler has missed the dues. Finally, the processor emulator calls the timer interrupt handler 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.

3.2.1.3 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.

3.2.1.4 *Dealing with Idle State.*

When all tasks are blocked and there is no task to run in a μITRON kernel, the kernel falls in to 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.

3.2.2 *TL4 Microkernel*

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

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.

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 in to 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 *3.2.1.4*).

3.3 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.

3.3.1 *Memory Footprints*

Table 1 shows the memory sizes consumed to run a single 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.

3.3.2 *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 3 (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 tasks, 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 1: Memory Foot Prints

	Text	Data	BSS
TL4 microkernel	23KB	4KB	20KB
μITRON kernel modified to run on *TL4* microkernel	18KB	1KB	44KB
Total	42KB	5KB	64KB
Original μITRON kernel	17KB	1kB	47KB

Table 2: Latencies from the Interrupt Processing Entry Point for a Serial Line Device

	μITRON on *TL4*	Original μITRON
T1: μITRON Interrupt Handler	2.45 μsec	3.75 μsec
T2: Character Received by Application Task 1	8.77 μsec	9.83 μsec
T3: Character Received by Application Task 2	10.49 μsec	10.72 μsec

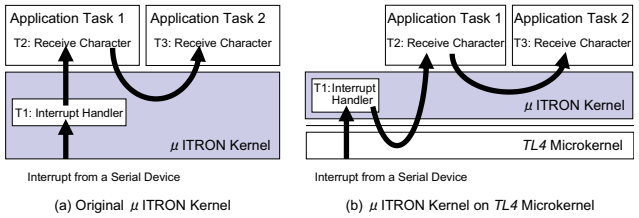

Figure 3: Measurement Setup to Handle Interrupts and to Invoke Application Tasks

Table 2 shows the measurements results. The table shows T1, T2, and T3 described above for the 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.

3.4 Current Status

We are currently extends the system to execute both Linux and μITRON on the TL4 microkernel. The system will support a resource management framework and device driver framework for running multiple operating systems in a stable way. The operating system also makes it possible to execute both Linux and μITRON. If an application requires very fast response, μITRON can be used to execute the applications. On the other hand, if an application requires to reuse various internet protocols, the application should be executed on Linux.

We are also implementing light-weight protection domains for confining software bugs in respective components. Since the component framework can be restarted by rebooting independently, the entire system does not need to be restarted when some serious problems are found. We have a plan to build a new operating system on the component framework. The operating system consists of a couple of components such as a file system component and a network system component. The components are quickly rebootable when the component is crashed. The operating system enables us to build a rebootable system that can reboot very quickly when a system finds a serious problem.

4. A LINUX-BASED OPERATING SYSTEM FOR BUILDING ROBUST EMBEDDED SYSTEMS

4.1 Introduction

The current Linux has several problems to be used for future embedded systems. When malicious application programs are downloaded and executed, the programs may consume a large amount of CPU capacity if those programs create many processes because these processes receive the CPU capacity fairly on the time-sharing scheduler.

Linux offers both the time-sharing scheduler and real-time scheduler. If there are real-time processes that are executed on the real-time scheduler, processes running on the time-

sharing scheduler are delayed until the real-time processes are blocked. Therefore, the response time of processes running on the time-sharing scheduler may become long due to the long execution of real-time processes.

We believe that the CPU resource should be protected from malicious programs, and the response time should not be degraded even if there are real-time processes. We have developed an accounting system to restrict the CPU capacity consumed by each process to solve the above problems.

4.2 Related Work

Linux/RK[7] provides the resource reservation system that reserves resources such as CPU, disk and network capacity for executing real-time applications. The resource reservation system prevents a system from being overloaded by controlling the admission of new processes. Also, the reservation system monitors that a process does not consume more CPU capacity than the specified CPU usage. The approach is very attractive for building real-time systems. Our accountingsystem uses a similar mechanism to restrict to use the CPU capacity to protect a system from malicious programs and long running real-time processes.

The proportional fair scheduler[1] allocates CPU capacity to each process fairly. The allocation of the CPU capacity is controlled in a rigorous way. Therefore, each process can consume the same amount of CPU resource. Each process can assign "weight" to allocate the different amount of CPU resource. For example, if process A has weight 1 and process B has weight 3, process A consumes 25% CPU capacity and process B consumes 75% CPU capacity. However, the approach is difficult to group processes to allocate CPU resource. To solve the problem, hierarchical proportional fair schedulers[3] are proposed. In the approach, the higher level multiple schedulers run on the lower level proportional fair scheduler. Each higher level scheduler consumes the same amount of CPU capacity. For example, two schedulers, scheduler A and B run on the proportional fair scheduler, and process X, Y and Z run on scheduler A. In this case, process X, Y and Z consume 50% CPU capacity.

The proportional fair scheduler is very promising, but it offers different semantics from the original Linux scheduling semantics. Some Linux applications assume the Linux's original scheduling semantics. Thus, the portability of applications is decreased in the approach.

4.3 Design and Implementation of Accounting System

This section proposes an accounting system to avoid unlimited use of the CPU capacity. We have implemented the system on Linux version 2.4, and the system runs on a standard PC.

4.3.1 Accounting Objects

The accounting system offers accounting objects as an abstraction to manipulate the CPU capacity of each process. An accounting object is bound to several processes to limit the CPU capacity that the processes can consume as shown in Figure 4.

As shown in Figure 5, an accounting object has two parameters: C and T, where T represents a period that is a constant time to control the object, and C is the maximum time to be able to execute processes bound to the accounting object within T. The processes bound to the accounting

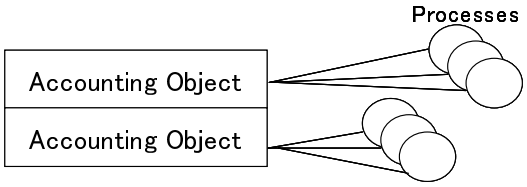

Figure 4: Accounting Object and Processes

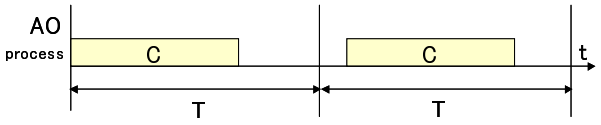

Figure 5: Parameters for Accounting Objects

object cannot consume the CPU time more than C within T. When processes consumes the entire CPU time within each period, the processes is blocked until the next period comes.

The accounting system allows several processes to be bound to an accounting object. For example, we assume that process P1, P2 and P3 are bound to an accounting object whose parameters are C1 and T1. When the total execution time of P1, P2 and P3 exceeds C1 in each T1, the execution of the processes are blocked until the next period will be started.

4.3.2 Binding Accounting Objects

An accounting object is bound to a process by invoking a system call to bind the accounting object and the process in an explicit way. Also, when a process that is bound to an accounting object executes the fork system call to create a new process, the accounting object is bound to the newly created process automatically in an implicit way.

4.3.3 Implementation

Figure 6 depicts the implementation of the accounting system. The current accounting system uses two types of timers for managing the CPU capacity that each process can consume. The first timer is a replenish timer and the second one is an enforcing timer for controlling each accounting object. Each accounting object has a value that shows the remaining execution time in each period. The value is decreased while the processes bound to the accounting object is running. When the value becomes 0, the processes are blocked until the next period starts. The replenish timer is used to set the timer value to parameter C in the accounting object, and the enforcing timer is used to check the value becomes 0.

In the default mode, the processes bound to an accounting object is blocked when the value becomes 0. However, if a process sets a signal number in parameters when creating the accounting object, the accounting system sends a signal instead of blocking it. The signal can be used to control the process bound to the accounting object in an explicit way.

4.3.4 Controlling Overload Situation

Controlling overload situation is very important to offer

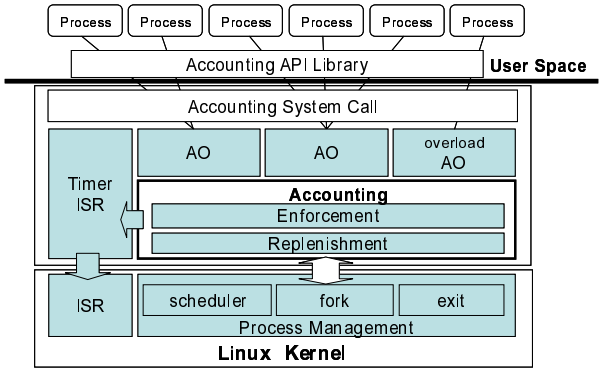

Figure 6: Accounting System

services in a stable way. In our system, an accounting object that is bound to the idle process is used to control overload situation. The idle process is executed when there is no runnable process. If the system becomes busy, the idle process consumes a very little CPU capacity. By monitoring the CPU usage of the idle process, the system can detect overload situation. Our system allows an application to receive a signal when the CPU usage of the idle process becomes lower than the specified CPU usage.

4.3.5 High Resolution Timer

The standard Linux kernel uses the 10 ms for the timer interrupt interval. Thus, it is meaningless to specify parameter C and T that are smaller than 10 ms. Our system solves the problem by using the high resolution timer[4] to support fine grained CPU accounting. The high resolution timer is implemented by using the one shot mode of the ISA clock timer chip in the standard PC. The high resolution timer allows us to specify microseconds granularity parameters for accounting objects.

4.3.6 Access Control

Each accounting object has an owner attribute for controlling the access to the object. The owner attribute of the accounting object is assigned by a process that creates it. Only the owner of an accountingobject or the super user can manipulate the accounting object. In previous resource reservation systems, there is no support of access control. Therefore, these previous systems cannot be used for protecting resources from malicious programs.

4.3.7 Kernel Interface

This section shows the kernel interface offered by the accounting system. An application program needs to use the *object_attribute* structure that contains basic parameters such as C and T for creating an accounting object.

int cabi_account_create (&object_id. &object_attributes) Creates a new accounting object. When created, the object is not bound to any processes. An application program sets parameters such as C and T in *object_attribute*. The function returns *object_id* that is used to manipulate the accounting object.

int cabi_account_destroy (object_id) Destroys the accounting objectspecified by *object_id*. The function can be called by the owner of the accounting object or the super user.

int cabi_account_bind_pid(object_id, pid) Binds the specified accounting object to a process whose process ID is pid. An accounting object can be bound to multiple processes. If the process is terminated, the process is automatically unbound from the accounting object.

int cabi_account_bind_pgid(object_id, pid) Binds all processes that have the same pgid to the specified accounting object.

int cabi_account_get (object_id, &object_attributes) Retrieves the parameters of the specified accounting object.

int cabi_account_object_set (object_id, &object_attributes) Changes the parameters of the specified accounting object.

4.3.8 An Example of Resource Protection

This section shows how our accounting system protects CPU resource from downloaded programs that behave maliciously. We assume that there is a manager process to start processes to execute downloaded programs. The manager process is bound to an accounting object whose C is 50 ms and T is 100 ms. When the manager process receives a request to execute a downloaded application, it creates a new process. The process is automatically bound to the accounting object to which the manager process is bound. Therefore, the newly created process cannot consume more than 50% CPU capacity(50 ms / 100 ms = 50 %), and it cannot monopolize the CPU resource.

4.3.9 An Example to Improve the Response Time

The accounting system can improve the response time of usual processes running on the time sharing scheduler. We assume that a real-time process is running and the process is executed for 500 ms without blocking. When a process that runs on the time sharing scheduler becomes runnable, the process may need to wait for the blocking of the real-time process. Therefore, the worst case response time becomes 500 ms. However, if an accounting object whose C is 40 ms and T is 50 ms is bound to the real-time process, the process is blocked every 50 ms for 10 ms. Thus, the worst case response time is improved to 40 ms by using our accounting system.

4.4 Evaluation

We have evaluated our accounting system by running several benchmarks. The evaluation uses a standard PC that has Celeron 300MHz CPU and 512MB memory. We use the high resolution timestamp counter in our evaluation.

4.4.1 Basic Cost

The evaluation shows the basic cost of our accounting system. Figure 7 shows the average cost for invoking each kernel function.

The result shows the creation time is longer than other system function's cost. Because the creation cost includes allocating a memory region and creating dynamic timers. The bind function also includes the searching cost through all processes in the system.

API	User (μ sec)
create	39.3
bind	35.1
unbind	9.8
destroy	24.0
set	4.9
get	3.5

Figure 7: Basic Overhead

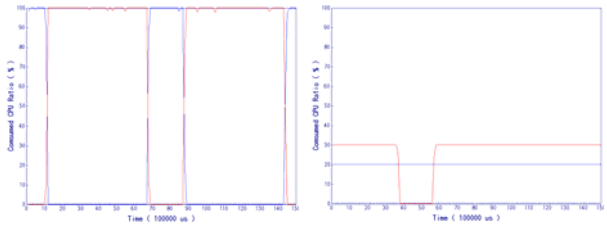

Figure 8: Improving the Response Time

4.4.2 *Improving the Response Time*

In the benchmark, we executed two tests as shown in Figure 8. The left graph in the figure shows the CPU usage of the two processes that are not bound to accounting objects. The first process executes an infinite loop and runs on the time sharing scheduler. The second process runs on the real-time scheduler, and stops 2 seconds after the execution of every 10 million loops.

The right graph shows the result when the second real-time process is bound to an accounting object whose C is 20 ms and T is 100 ms. By binding to an accounting object, a process running on the time sharing scheduler can consume CPU capacity constantly. Thus, the response time is improved significantly.

4.4.3 *Protecting CPU Resource*

In the benchmark, a process that simulates a malicious program creates many processes to monopolize the entire CPU resource. The process is bound to an accounting object whose C is 30 ms and T is 100 ms. When the process creates processes, these processes are bound to the parent process's accounting object automatically.

Figure 9 shows the CPU usage when malicious processes are executed. The result shows that these malicious processes can consume only 30% CPU capacity, and cannot monopolize the entire CPU resource.

4.5 Current Status

We have developed a prototype accountingsystem and shown the effectiveness of our approach. Currently, Montavista Japan Inc. is working to merge our system in their embedded Linux distribution, and Japan Embedded Linux Consortium started a new working group to discuss the kernel interface of the accounting system for various embedded systems. We are currently considering to enhance our system for supporting multiple security domains and multi-core chips. Also, we need to take into account other resources

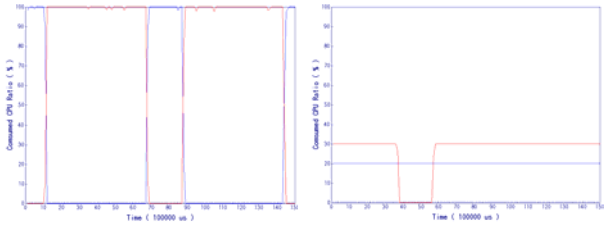

Figure 9: Protecting from a Malicious Application

such as memory , disk bandwidth, and network bandwidth in the near future.

5. CONCLUSION

In this paper, we have presented two operating systems to increase the robustness of embedded systems. The first operating system is a microkernel-based operating system that enables multiple μITRON to be executed on a microkernel. The second operating system enhances embedded Linux to avoid to monopolize the entire CPU capacity from malicious applications.

6. REFERENCES

[1] A.Demers, S.Keshav, and S. Shenker, Analysis and simulation of a fair queueing algorithm, Proceedings of ACM SIGCOMM 1989, pp.1-12, September 1989.

[2] H.-W.Gellersen. A.Schmidt, and M.Beigl, "Adding Some Smartness to Devices and Everyday Things", In Proceedings of the Third Workshop on Mobile Computing System and Applications, 2000.

[3] Pawan Goyal, Xingang Guo, and Harrick M. Vin,A Hierarchical CPU Scheduler for Multimedia Operating Systems, A Hierarchical CPU Scheduler for Multimedia Operating Systems, 1996.

[4] High Resolution Timer, http://high-res-timers.sourceforge.net/

[5] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.

[6] T. Nakajima, et. al., "Technology Challenges for Building Internet-Scale Ubiquitous Computing", In Proceedings of the Seventh IEEE International Workshop on Object-oriented Real-time Dependable Systems, 2002.

[7] Shuichi Oikawa and Ragunathan Rajkumar, Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior, In Proceedings of the IEEE Real-Time Technology and Applications Symposium Vancouver, June 1999.

[8] David C. Steere, Ashvin Goel, Joshua Gruenbeerg, Dylan McNamee, Calton Pu, Jonathan Walpole, Feedback-driven Proportion Allocator for Real-Rate Scheduling, Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, Louisiana, February, 1999

[9] Hiroaki Takada ed. μITRON4.0 Specification. TRON Association, 1999. (In Japanese)