ARX/ULTRA: A New Real-Time Kernel Architecture for Supporting User-Level Threads *

Seongsoo Hong, Yangmin Seo, Jungkeun Park†

Abstract

In this paper we present a new real-time kernel architecture that is capable of supporting user-level threads. Although user-level threads offer an essential mechanism for structuring applications and representing concurrency, they are not widely exploited in real-time programming for the lack of capabilities for signal handling and thread scheduling. The kernel, we named ARX, provides a clean and efficient solution to this problem. First, it provides *dynamic stack binding* to prevent a blocked thread from preempting the entire process. Second, it offers the *scheduling event upcall* facility to efficiently propagate kernel events to users so that the user-level scheduler can timely handle events in an application-specific way. In order to reduce the upcall overhead, the kernel also provides a novel, lock-free implementation of thread schedulers. Finally, the kernel offers a reservation-based process scheduling mechanism for real-time processes.

We have successfully implemented full-fledged real-time kernel *ARX* and user-level thread package *ULTRA* on an Intel Pentium PC. We have ported representative soft and hard real-time applications onto the ARX kernel, and performed extensive experiments. Our experiments clearly show that the ARX kernel mechanisms incur very low overhead and offer acceptable performance with real-time guarantees.

*The work reported in this paper was supported in part by Engineering Research Center for Advanced Control and Instrumentation (ERC-ACI) under Grant 96K3-0707-02-06-1, KOSEF under Grant 96-2037, and S.N.U. Korea Electric Power Corp. Research Fund under Grant 96-15-1135.

†School of Electrical Engineering and ERC-ACI, Seoul National University, Seoul 151-742, Korea. Email: sshong@redwood.snu.ac.kr.

1 Introduction

In modern operating systems, threads have emerged as an essential mechanism for structuring applications and representing concurrency [1]. They are extensively substituting traditional UNIX-like processes. More and more operating systems have been modified, and even created to support threaded programming. For example, Sun Microsystems extended their own operating system to incorporate multithreading capability [15, 17]; many micro-kernel based operating systems including Mach and Windows NT were designed from scratch to support multithreading [3, 19]. This trend is not an exception in real-time operating systems. POSIX specifies a priority-driven thread model in its thread extension standard [5].

The reason behind such popularity is clear. Not only threads can offer the desired level of abstraction for representing concurrency, but also it can support efficient resource management and fast context switching. This helps real-time operating systems provide short and bounded dispatch latency for real-time threads. Moreover, threads allow programmers to structure programs such that multiple asynchronous events in the programs can be effectively handled.

Threads can be implemented in various ways and incorporated into different places in a programming environment [12]. First, they can be implemented as a set of kernel functionalities. In this implementation, the kernel can recognize individual threads and provide scheduling for them. This implementation is often called *kernel-level* threads. Second, threads can be implemented as a set of library functions. In this implementation, the kernel cannot directly see threads, since they are built on top of traditional processes. This implementation is referred to as *user-level* threads. Finally, threads can be implemented as a mixture of the above two.

Obviously, there is a tradeoff involved between the kernel-level implementation and the user-level implementation. The kernel-level threads offer more efficient scheduling, but less flexible abstraction than the user-level threads do. This motivates us to investigate a new kernel architecture which best optimizes this tradeoff for real-time threads.

1.1 Motivations

In conventional programming, user-level threads are much more favored than kernel-level threads mainly for two reasons [1]: (1) kernel-level threads provide wrong abstraction for a programming environment, since they require that every application should be mapped onto a single thread implementation that the kernel provides; and (2) kernel-level threads incur more overhead than user-level threads, since with the kernel-level threads, processes must cross an extra user/kernel protection boundary for each thread operation.

On the other hand, in real-time programming, kernel-level threads are more often the choice of implementation than user-level threads, albeit those disadvantages of the kernel-level threads. The kernel-level implementation can simplify thread scheduling and signal handling, which is a very critical property for real-time processing. With no kernel support, the user-level threads cannot deal with kernel-level thread blocking and signal propagation. Thus, when a user-level thread is blocked in the kernel mode via a blocking system call, the entire process of the thread is blocked along with many other runnable threads in that process.

Sun Threads took the kernel-level thread implementation for real-time processing in that each real-time thread is associated with a kernel thread [15, 17]. In such an implementation, all real-time threads, regardless of processes to which they belong, form a flat task space which is consistent with the task model of preemptive priority scheduling.

However, such a simplified scheduling mechanism seriously restricts the application model. Since real-time threads in all processes (applications) are subject to a single scheduling policy, it is very difficult, though not impossible, to provide application-specific scheduling support for threads in different processes. For example, consider an automated manufacturing system where a single workstation has to simultaneously run two applications: one that monitors a manufacturing cell via a digital camera, and the other controlling a numerical control (NC) machine. The digital video application may be efficiently run with EDF scheduling, whereas the NC machine controller may use cyclic executive scheduling [2, 10] for deterministic responses. Such a system, however, cannot be implemented with kernel-level threads, since the two scheduling policies bear no similarity: cyclic executive scheduling is an off-line scheduling policy in which thread schedules are computed and fixed prior to run-time, while EDF scheduling is a priority-based policy in which the thread priorities are determined at run-time. Unless we modify the kernel scheduler reflecting the specific requirement of each application, we cannot make the kernel support two distinct scheduling policies. As a result, the inflexibility of the kernel-level threads compromises an important feature of threads – a versatile abstraction for representing concurrency.

In this paper, we propose a new architecture for a real-time kernel that is capable of supporting user-level threads. We refer to this architecture as *ARX* in which user-level threads are reliably supported, and the real-time requirements are safely guaranteed, based on CPU capacity reservation made for each process. In the ARX approach, each real-time process is given a share of the CPU according to its weight, and threads in the process are scheduled to consume the share via a user-level scheduler.

We have implemented a full-fledged kernel and a user-level thread package called *ULTRA* (*User-Level Threads for Real-time Applications*), on an Intel Pentium PC. We have also ported an MPEG player and a CNC (computerized numerical control) controller program on our kernel, and performed extensive experiments with different mixes of soft and hard real-time applications. Our experiments clearly demonstrate that the ARX kernel mechanisms incur very low overhead and offer acceptable performance with deterministic guarantees. We report the results of these experiments.

1.2 Related Work

The ARX approach is built upon previous work on thread implementation and scheduling. In particular, we adopted several techniques presented in [1, 6] and extended them, since they presented only partial solutions to our problem. First, in [6], Govindan and Anderson proposed the split-level scheduling of real-time threads for continuous media applications. In this approach, the virtual address space of a process contains multiple real-time threads called lightweight processes (LWPs), and they are scheduled via a deadline-based scheduling policy such as EDF scheduling. A thread scheduler is partitioned into a user-level component and a kernel-level component which communicate with each other via shared memory. The user-level scheduler of a process notifies the kernel of the nearest deadline in the process. The kernel, in turn, determines the thread with the shortest deadline in the system, and informs all user-level schedulers of the deadline so that the currently running process can voluntarily give up the CPU by the deadline.

This is the first work that we know of, to support real-time applications only with user-level threads. In fact, this scheme was implemented in SunOS 4.1 which did not support kernel-level threads. However, this approach has two drawbacks. One is a well-known thread implementation problem which occurs when a thread is blocked in the kernel mode. If a thread of a process is blocked in the kernel mode, the kernel takes away the CPU from it and gives the CPU to another process, even though the process has runnable threads. The other problem is in tight coupling between user-level and kernel-level schedulers. Such a coupling makes it difficult to employ multiple scheduling policies in the system for applications with different requirements. It also makes it difficult to maintain separation of concerns between applications, since threads in distinct address spaces can interfere with each other. Consequently, this coupling becomes the major drawback of the split-level scheduling approach.

In [1], Anderson *et al.* proposed a new thread implementation mechanism to effectively manage user-level parallelism. They pointed out that the performance of kernel threads was inherently worse than that of user-level threads, and that the problems encountered in integrating user-level threads with other system services were the consequence of the lack of kernel support for user-level threads. In order to maximize user-level parallelism, they proposed a scheduler activation which transfers control from the kernel to a user-level scheduler, upon the occurrence of a kernel event. The user-level scheduler can use an activation to modify user-level thread data structures, execute user-level threads, and make requests of the kernel.

Though the scheduler activation approach gives a clean solution for the management of user-level parallelism, it is not best suited to real-time processing. In this approach, the execution of high priority threads can be unnecessarily delayed, since the kernel let a resumed thread finish the remainder of its system call before it yields the CPU. Moreover, a user-level scheduler cannot support real-time scheduling, since timer interrupts are handled in the kernel mode and they are not propagated to the user level. For real-time processing, we specialize the scheduler activation mechanism, and refer to the new scheme as *dynamic stack binding*. Unlike the original approach, our scheme arranges a user-level scheduler to run on its own stack, still enabling the kernel to dynamically create and bind kernel stacks to user-level threads. Such a scheme allows for a much lighter implementation and faster real-time responses than the scheduler activation does. However, since multiple instances of a user-level scheduler can compete for a single stack, we devise a lock-free implementation of a reentrant user-level scheduler.

The ARX approach is also inspired by the recent advances of QoS (quality of service) scheduling techniques. Specifically, it relies on the weighted fair queuing (WFQ) algorithm [4, 13], for the kernel-level process scheduling. Although the WFQ algorithm was devised for proportional bandwidth allocation in communication networks, it works for CPU scheduling, as shown in [7, 18]. The algorithm emulates a hypothetical weighted round robin server where in each round a packet receives service that is infinitesimal and proportional to the weight of the packet [7]. It achieves weighted fairness by introducing virtual time which advances at a rate inversely proportional to the aggregated weight of the back-logged processes. Processes are assigned virtual finish times on arrival, and they are scheduled in the increasing order of virtual finish times.

There are several variants of the WFQ algorithm. Goyal *et al.* proposed the start-time fair queuing (SFQ) algorithm in [8] to achieve weighted fairness regardless of variation in the server capacity. They applied the SFQ algorithm to CPU scheduling in [7]. Unlike the original algorithm, the SFQ algorithm tags virtual start times to the arrivals of processes and schedules them in the increasing order of start tags. This algorithm requires less computation to track the virtual time than other variants of the WFQ algorithm. However, we found out that the SFQ algorithm yielded a serious problem for real-time processing, when we investigated its use in our kernel; that is, the SFQ algorithm may not be able to provide fairness. When a process gets idle before the subsequent request arrival, it has to yield the CPU to other process, and thus loses some of its share. This happens because processes are scheduled based on their start tags, but they are computed without considering the weights of the processes.

1.3 Remainder of the Paper

This paper is organized as follows. In Section 2, we discuss the process and thread models we use in the ARX approach. We then show the design goals we possessed during the design of our kernel. We also give the overview of the ARX architecture as a solution to these goals. Section 3 gives an in-depth discussion of the ARX kernel covering the kernel scheduler, the event upcall facility, and the user-level scheduler. Section 4 discusses the ARX/ULTRA implementation. In Section 5, we report the results of our experiments with the kernel. Finally, we conclude the paper with future research directions and our contributions in Section 6.

2 Design Goals and Kernel Overview

In this section, we examine the process and thread models that we use in the ARX approach. We then discuss the design goals we possessed during the design of the ARX kernel and present the ARX architecture as a solution to these goals.

2.1 Process and Thread Models

As in many traditional multithreading systems, a process (application) in our approach consists of two components: a set of threads and a collection of resources assigned to the process. A thread is a dynamic entity that denotes a control point in the process and that executes a sequence of instructions. Resources constitute an execution environment for threads, which includes an address space, I/O devices, common data structures, etc. These resources are shared by all the threads in the process. In addition, each thread has its own private resources such as a program counter, a stack, and a register context. In the ARX approach, thread abstraction is given entirely in the user level, and the kernel is not aware of user-level threads. This is accomplished through a thread library package and application-specific user-level thread schedulers. The thread library package includes functions for creating, synchronizing, and scheduling threads. The execution of these functions does not require kernel services. The user-level thread scheduler interacts with the kernel via well-defined interfaces such as system calls and scheduling event upcalls.

The ARX kernel offers two types of classes for processes, namely a time-critical (real-time) class and a conventional time-sharing class. Processes in a real-time class are preferred to those in a time-sharing class. Unlike in other operating systems, real-time processes do not have priorities; instead, they are scheduled based on their shares of CPU bandwidth. Programmers can negotiate with the kernel about a share, when a process is created, or executed. A real-time process may possess many periodic and aperiodic threads. It is programmers' responsibility to translate the timing constraints of real-time threads into the CPU bandwidth. We will get back to this issue in Section 6.

2.2 Design Goals and Our Solutions

When we designed a real-time kernel that can support user-level threads, we possessed the following design goals.

(1) The kernel scheduler must provide flexible and predictable scheduling services for various real-time applications.

(2) A thread blocked in the kernel mode must not affect the execution of other threads in the same process.

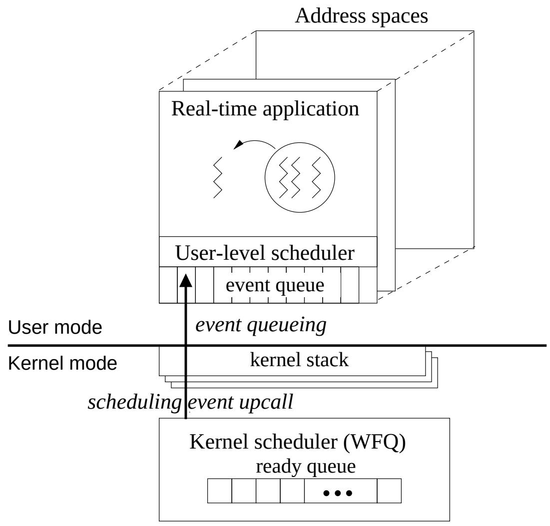


Figure 1: The overall structure of the ARX kernel.

(3) The user-level scheduler must be able to efficiently handle scheduling events including timer interrupts.

For the first goal, we required that the kernel scheduler should not be involved in thread scheduling, but provide an address space switching mechanism that can maintain fairness among real-time processes. A class of scheduling algorithms that meet this requirement includes the weighted fair queuing (WFQ) algorithm [4, 13] and its variants [8, 18]. We incorporated the WFQ algorithm in the ARX kernel and used it for process scheduling.

For the second goal, we adopted the scheduler activation from [1] and specialized it for real-time processing. We refer to the new scheme as *dynamic stack binding*. With this scheme, the kernel can instantly create a kernel stack and bind it to another ready thread.

Finally, for the third goal, we devised a mechanism we named *scheduling event upcall*. With this scheme, the ARX kernel can asynchronously invoke a user-level scheduler in response to a scheduling event. The invoked scheduler can in turn take care of the event in an application-specific way. Since scheduling event upcalls are frequently made, we also proposed a novel, lock-free implementation of event queues, in order to reduce the upcall overhead.

Figure 1 shows the overall structure of the ARX kernel. The kernel maintains a queue of processes and performs process scheduling based on the WFQ algorithm. It also dynamically creates and destroys kernel stacks so that a ready thread cannot be affected by a blocked or preempted thread. In each address space, there exists a user-level scheduler which maintains the queue of scheduling events. The kernel enqueues an event into the queue during an upcall, and the corresponding user-level scheduler dequeues it right after the upcall.

3 The ARX Kernel Architecture

In this section, we give a detailed discussion of the ARX kernel architecture. We first describe the kernel scheduler with the discussion of the WFQ algorithm. We then present the event upcall facility along with several optimization techniques devised to reduce the upcall overhead. Finally, we present a novel, lock-free implementation of user-level schedulers that can resolve the consistency problem of shared data structures.

3.1 The Kernel Scheduler

The ARX kernel makes use of the WFQ algorithm for process scheduling. The algorithm was devised by Demers, Keshav, and Shenker in [4] for bandwidth allocation in communication networks. It was originally intended to emulate a hypothetical weighted round robin server where in each round a packet receives service that is infinitesimal and proportional to the weight of the packet. Though it was designed for communication networks, it works for CPU scheduling, as shown in [7, 18].

Conceptually, the WFQ algorithm partitions a server into multiple hypothetical ones whose capacities are proportional to the relative weights of the clients. For example, given three applications with weights of 1, 2, and 3, the algorithm divides the server into three whose capacities are 1/6, 1/3, and 1/2, respectively. However, in practice, packet transmission or process execution on a CPU cannot be arbitrarily preempted, and thus the service is often made on a quantum basis. Such quantization of a service unit may result in unfairness in a real server. To assuage this problem, the WFQ algorithm introduces the notion of virtual time and assigns virtual start and finish times to processes. It schedules processes in the increasing order of their virtual finish times.

The virtual time $V(t)$ at real time t denotes a round number that is in progress at time t in the hypothetical round robin server. More specifically, $V(t)$ is defined as below.

$$\frac{dV(t)}{dt} = \frac{1}{\sum_{k \in B(t)} w_k} \tag{1}$$

where w_i is the integer weight of process τ_i, $B(t)$ is the back-logged set of processes at time t, and $V(0) = 0$ initially. Note that $B(t)$ is changed, only when a process joins or leaves the competition with other processes. The virtual time advances at a rate inversely proportional to the aggregated weight of the set $B(t)$.

Consider a process τ_i which joins the competition at time t_1, and leaves it at time t_2. Then we get the virtual time at t_2 as follows.

$$V(t_2) = V(t_1) + \int_{t_1}^{t_2} \frac{1}{\sum_{k \in B(t)} w_k} dt \tag{2}$$

Suppose that the kth quantum of process τ_i is requested at time a_i^k. Then the virtual start time

S_i^k and the virtual finish time F_i^k of τ_i are defined as follows.

$$S_i^k = \max\{V(a_i^k), F_i^{k-1}\} \quad k \geq 1$$

$$F_i^k = S_i^k + \frac{C_i^k}{w_i} \quad k \geq 1 \tag{3}$$

where C_i^k is the execution time of the request, and $F_i^0 = 0$ initially.

With the WFQ algorithm, we can bound the maximum delay that a process can experience. In [4], the maximum delay bound of process τ_i is given as below.

$$EAT(a_i^k) + \frac{C_i^k}{r_i} + C^{max}$$

where $EAT(a_i^k)$ is the kth expected arrival time of process τ_i and C^{max} is the maximum execution time of a quantum. Note that the process τ_i will get the relative share $r_i = \frac{w_i}{\sum_{j \in A(t)} w_j}$ where $A(t)$ is the set of active processes at time t. Since r_i is inversely proportional to the load of the active processes, the delay bound increases with the number of applications. This renders it difficult to use the WFQ algorithm for hard real-time applications, since the response time of a hard real-time process may be affected by the arrival of a new application. As an engineering solution to this problem, we restrict that the total weight of all the real-time applications should not exceed a specified bound. Note that processes in a time-sharing class do not interfere with real-time processes.

We have implemented the WFQ scheduler in the ARX kernel. A real-time process can request for CPU bandwidth at creation time and at run time via `fork()` and `qosctrl()` system calls, respectively.

3.2 Scheduling Event Upcall

The biggest problem in supporting real-time threads with a pure user-level thread implementation is the lack of facilities to handle kernel events that may affect thread scheduling. In most traditional operating systems, the kernel events are caught and handled in the kernel. This does not pose a problem in those operating systems where the kernel performs process scheduling. On the other hand, in the user-level thread implementation where thread schedulers are required to run in the user mode, kernel-level event handling leads to a problem, since thread schedulers cannot effectively interrupt running threads without kernel support. This makes it extremely difficult to implement preemptive scheduling in the user-level scheduler, and often restricts the utility of user-level threads for real-time processing.

In the ARX approach, we propose a novel scheme which allows the kernel to efficiently propagate kernel events to user-level schedulers. We refer to this scheme as *scheduling event upcall.* We use the term "upcall" to contrast it with a system call, since an upcall is made in an opposite direction – from the kernel to a user – to a system call. Since upcalls incur an extra overhead to the system, it is important to reduce their cost. To do so, we make two optimizations: (1) we arrange each user-level scheduler to inform the kernel of only those events that it is interested in so as to prevent the kernel from making irrelevant upcalls; and (2) we devise a novel, lock-free implementation of event queues so as to reduce blocking time during an upcall. In the remainder of this section, we discuss these optimizations.

We first show events that cause upcalls.

- `BLOCK`: When a thread is blocked in the kernel, the kernel sends the user-level scheduler a `BLOCK` event. The event contains the kernel stack id of the blocked thread.
- `WAKEUP`: When a blocked thread wakes up in the kernel, the kernel sends a `WAKEUP` event. The event contains the kernel stack id of the woken-up thread.
- `RESUMPTION`: When a process gets the CPU, the kernel awakes the user-level scheduler of the resuming process by sending it a `RESUMPTION` event.
- `TIMER`: When the timer expires at time preset by a user-level scheduler, the kernel sends a `TIMER` event to that particular scheduler. Upon receiving a `TIMER` upcall, a scheduler takes away the CPU from the current thread and gives it to another ready thread.

Observe that the ARX kernel enables user-level schedulers to mark the next rescheduling point – or the occurrence time of the upcoming scheduling event – in the system. Such an event may be the arrival of the next periodic thread in a priority-based scheduler, or the start of the next minor cycle in a cyclic executive. When time is up at this rescheduling point, the kernel makes a `TIMER` upcall. Such a reservation mechanism can help reduce the number of upcalls, since the kernel need not make an upcall, every time the real-time clock ticks. This is one of the distinguished and essential features of the ARX kernel.

We also make further optimization for scheduling event upcalls. When there exist multiple events pending to be notified, the ARX kernel merges them into a single upcall. As an example, consider a case where a timer interrupt goes off right after an I/O completion event. (We assume that a timer interrupt has priority over an I/O interrupt.) As a result, a timer interrupt handler preempts the I/O handler and prepares a `TIMER` upcall. Returning from the interrupt, it finds out that another event (I/O interrupt) is pending. Instead of making the upcall right away, it jumps to the I/O interrupt handler. The I/O interrupt handler in turn executes the wakeup procedure and merges the `WAKEUP` event with the `TIMER` event, and finally makes a merged upcall to the user-level scheduler.

When the kernel performs an upcall for an event, it enqueues the event into a *scheduler control block* which is mapped into an application's address space. The scheduler control block serves for two roles: (1) the execution stack for a user-level scheduler, and (2) a communication channel

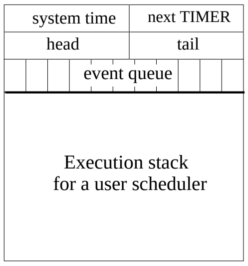


Figure 2: The scheduler control block.

between a user-level scheduler and the kernel. Figure 2 shows the structure of the scheduler control block. It is partitioned into two subsections. The upper subsection holds an event queue, and the process specific information such as the head and the tail of the event queue, current system time, and next rescheduling time. The lower subsection serves as an execution stack for the user-level scheduler. The current system time field of a running process in the system is updated by the kernel, whenever the real-time clock ticks. On the other hand, the next rescheduling time field of a process is updated by the user-level scheduler, when it receives a `TIMER` event: the user-level scheduler first determines the arrival time of the next periodic thread based on the system time specified in the scheduler control block, and then writes the next rescheduling time into the scheduler control block. This scheme can help reduce the upcall overhead, since the user-level scheduler need not make a system call to communicate with the kernel.

An enqueued event contains at least two pieces of information: an event type and the context of a thread running at the time of the upcall. The event type indicates which event caused the upcall. This field also contains a flag which indicates the machine state – the user mode or the kernel mode – at the time of the upcall. The thread context may be either a register context or a kernel context, depending on the flag. The kernel context contains the blocked kernel stack id.

3.3 Reentrant User-Level Schedulers

Another source of the upcall overhead is in the fact that the kernel can be blocked for consistency, when it attempts to access an event queue. In most conventional time-sharing operating systems, accesses to shared data structures are guarded by a pair of locking and unlocking primitives such as `sighold()` and `sigrelse()` system calls in UNIX. When the kernel attempts to access a shared data structure while signals are blocked, it should wait until signals are unblocked again. If we adopt this scheme, we must disable the kernel from making any upcalls while a user-level scheduler is executed. After all, the event queue is a shared data structure which is read by a user-level scheduler at one end, and modified by the kernel at the other.

However, this scheme has serious drawbacks. First, the execution of user-level schedulers always

```
fixed_priority_scheduler()
{
    if (locked by a library function)
        return;
    while (head != tail) {
        process upcall;
        save context to TCB;
        head = (head+1)%MAX_QUEUE;
    }

    select the highest priority thread from ready queue;
    if (no thread to run)
        yield();
    else {
        if (kernel mode)
            resume(kernel context structure id);
        else
            intra-level context switch;
    }
}
```

Figure 3: A user-level scheduler implementing fixed priority scheduling.

involves system calls, and thus the performance benefits of the user-level threads will completely vanish. Second, the kernel may lose pending events, if an event of the same type occurs while upcalls are being disabled. To deal with these problems, we devise a novel, lock-free implementation of a user-level scheduler in that the kernel is never refused to make an upcall, while the consistency of shared data structures, such as the event queues and thread control blocks, is always guaranteed.

In this scheme, all user-level schedulers, regardless of scheduling policies they implement, must be coded following the reference structure that the ARX kernel provides. In the reference structure, the code of a user-level scheduler is partitioned into two sections. The upper section of the code is a *dequeuer* which repeatedly takes out events from a queue, until it empties the queue. For each event, it performs event-specific actions such as priority adjustment in a dynamic priority scheduler, and copies the context information from the event to the thread control block of the corresponding thread. The lower section is a *dispatcher* which determines a thread to run next, and actually dispatches it.

As an illustration, in Figure 3, we show a user-level scheduler which implements a fixed priority scheduling policy. The first section repeatedly takes out an event, decodes its type, and inserts a corresponding thread into the ready queue or the sleep queue. The second section selects the highest priority thread from the ready queue and performs a context switch to it.

We design the dequeuer code of user-level schedulers based on the following observations.

- The kernel always writes into an event queue.
- A user-level scheduler always reads from an event queue.
- A user-level scheduler has its own private user stack.

- Thread preemption can be initiated only by an upcall from the kernel.

In our scheme, the kernel and user-level schedulers work according to the following rules.

(1) Whenever an event occurs, the kernel enqueues it at the end of the queue. When the queue holds more than one entry, then the kernel simply returns. Otherwise, it invokes the user-level scheduler.

(2) When invoked, user-level schedulers greedily process the events in the queue.

Note that the kernel can safely insert an event into a queue and a user-level scheduler can safely remove one from the queue without worrying about each other. They modify two different pointers, namely, the tail and the head pointers, respectively – unless one pointer crosses over the other due to an error. Since user-level schedulers greedily consume events, when the kernel sees a nonempty queue before enqueuing an event, there must exist one user-level scheduler which has already been preempted.

In the ARX kernel, thread scheduling can be initiated by either event upcalls or library calls to thread and lock management functions. Thus, it is possible that the kernel invokes a user-level scheduler via an upcall, while a thread library function accesses shared data structures. We resolve the consistency problem of this case by introducing a global lock variable. It simulates an interrupt disable and enable mechanism. When a library function enters a critical section, it sets the lock variable. When a user-level scheduler is invoked by the kernel while the variable is set, it does nothing and simply returns. Then, control is returned to a point in the library function. When the function finishes its critical section, it checks if there are pending upcalls. If so, the function saves its context for rescheduling, resets the lock variable, and then invokes the user-level scheduler. This can be done with no kernel support.

The consistency problem gets more complex, when the multiple instances of the same user-level scheduler are simultaneously invoked. We design the dispatcher code of user-level schedulers as follows.

- The task done by the dispatcher code can always be discarded without jeopardizing the consistency of the shared data structures.

To make this possible, we neither associate a thread control block with a user-level scheduler, nor save its context even if it gets preempted. As a result, a new instance of the same scheduler can safely wipe out the stack space of the old instance, and restart from the beginning.

It is not difficult to see such a design lead to a reentrant user-level scheduler which maintains the system consistency.

3.4 Rendering of Scheduler Execution

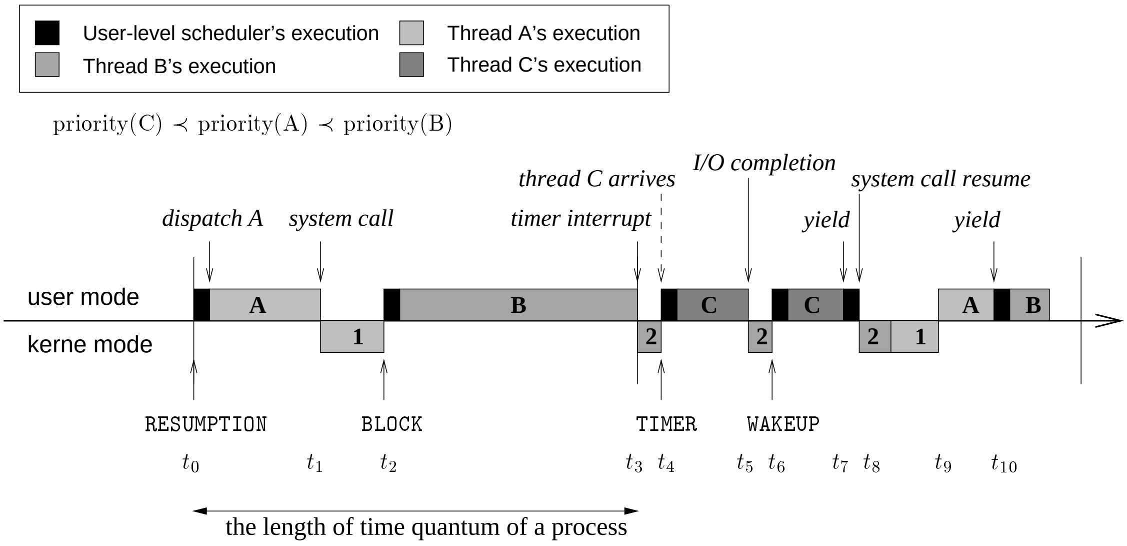


Figure 4: An execution scenario of three user-level threads.

As an illustration, we consider a real-time process which runs with fixed priority scheduling. The process possesses three threads A, B, and C where thread C has the highest priority and thread B the lowest. Figure 4 renders the executions of these threads. In the figure, the area above the timeline denotes the user mode, and the area below it denotes the kernel mode. As usual, system calls and interrupt handlers are executed in the kernel mode. The numbers in the kernel mode execution denote the ids of kernel stacks that are used for the time intervals. Suppose that threads A and B are ready at time t_0, and thread C becomes runnable at time t_4. As an initial condition, we assume that the first kernel stack id is 1, and that thread A and B are resumed in the user mode.

(1) First, at time t_0, the kernel resumes the process's user-level scheduler via a `RESUMPTION` event. Being invoked, the user-level scheduler selects thread A to be the highest priority thread, and dispatches it.

(2) At time t_1, thread A makes a system call to receive an I/O service. It runs in the kernel mode with the kernel stack of id 1 which was available for the process.

(3) At time t_2, the kernel initiates a DMA operation for the requested I/O, and then calls `sleep()` function to wait for an I/O completion event. The `sleep()` function inserts the process along with its process id and the kernel stack id into the sleep queue. It then creates a new kernel stack of id 2, switches the process's kernel stack with the new one, and finally makes a `BLOCK` upcall providing the blocked kernel stack id 1 as an upcall argument. Upon receiving the `BLOCK` event, the user-level scheduler records the blocked kernel stack id 1 in thread A's thread control block, and removes thread A from the ready queue. Finally, it dispatches

thread B which is the only runnable thread at that time.

(4) At time t_3, a timer interrupt handler is activated, and runs on the kernel stack 2. It checks whether to make a `TIMER` upcall by comparing the next rescheduling time specified in the scheduler control block with the system time. At time t_4, making a `TIMER` upcall, the kernel let the user-level scheduler know the arrival of thread C. As a result, thread C is put into the ready queue, and runs next, since it becomes the highest priority thread.

(5) At time t_5, the I/O interrupt goes off. The I/O interrupt handler is invoked, and performs wakeup operations. At time t_6, making a `WAKEUP` upcall with old kernel stack id 1 being its argument, the kernel let the user-level scheduler know that thread A has been woken up in the kernel mode with the kernel stack id 1. Thread A is put into the ready queue. Since thread C is still the highest priority thread, it continues it execution until t_7 when it calls `thread_sleep()` library function to relinquish the CPU. Note that with the scheduler activation [1], the control could not be given to thread C at t_6; instead, thread A would continue until it finishes up the blocked system call. This would lead to unnecessary priority inversion for thread C.

(6) At time t_8, the user-level scheduler picks thread A for execution, and finds out that the context of thread A was saved in the kernel mode. Since the user-level scheduler cannot immediately resume thread A in the kernel mode, it invokes `resume()` system call with kernel stack id 1 as an argument. The `resume()` system call then runs on the kernel stack 2. It then switches the current stack (id 2) with the old kernel stack (id 1), discards the current kernel stack (id 2), and restores the old context. Thread A executes the remainder of the blocked system call until t_9.

(7) At time t_9, thread A completes the system call, and returns to the user mode. Note that the kernel does not invoke the user-level scheduler at this point, since there is no kernel event that affects user-level scheduling.

(8) Finally, at time t_{10}, thread A relinquishes the CPU.

4 Implementation of the ARX Kernel

In order to demonstrate the viability of the ARX approach, we have built our own operating system which consists of a fully preemptive real-time kernel and a thread library package. Although we could have implemented the ARX mechanisms on top of other existing operating systems such as UNIX SVR4 and BSD UNIX, we decided to develop a new one, since they did not offer a fully preemptive kernel.

The ARX kernel reliably realizes weighted fair queuing, dynamic stack binding, and scheduling event upcall, and it is currently up and running on a Pentium PC. The kernel is composed of three components, namely, process, memory, and file management subsystems. It also includes a subset of Internet protocols such as UDP/IP to support TFTP and NFS during the kernel development. The memory management subsystem realizes the paging mechanism to avoid memory fragmentation and provide memory protections between address spaces. It exploits the paging architecture of the Intel 80386 processor. It adopts page pinning for real-time applications to avoid unpredictability due to page faults. As a result, all active real-time applications are guaranteed to be resident in the main memory. The file management subsystem implements the FAT file system of MS-DOS. The kernel also includes custom-made drivers for essential devices such as a console, the Ethernet cards, disk drives, etc.

As in many operating systems, the ARX kernel offers a variety of system calls for device I/O operations, communications, file accesses, and process scheduling. To summarize, we list below the key subset of the ARX system calls. On the other hand, services for thread scheduling, synchronization and IPCs are provided through thread library functions, rather than system calls.

- `int register_uscheduler(int uscheduler, int scb, int scb_size)`: An application invokes this system call to register its user-level scheduler, the base address, and the size of the scheduler control block. When this system call is invoked, the kernel partitions the scheduler control block into two subsections, as described in Section 3.2, and returns the top address of the user-level scheduler's stack.
- `int fork(int class, int weight)`: This system call replicates the entire states of the parent process including text and data segments. The first argument denotes a process class (time-sharing or real-time class), and the second a CPU share that the forked process will receive. The share ranges from 1 to 100.
- `int qosctrl(int class, int weight)`: An application invokes this system call to change its class and its share of the CPU capacity. The arguments are the same as those of the `fork()` system call.
- `int yield(long nuptime)`: When an application has no runnable threads, the user-level scheduler invokes this system call to voluntarily relinquish the CPU. The argument `nuptime` denotes the absolute time when it wants to restore the CPU. If it is zero and the process has no thread being blocked in the kernel mode, the process will never be runnable.
- `int resume(int kctx_id)`: The user-level scheduler invokes this system call to resume a thread which was blocked or preempted in the kernel mode.

We have also implemented the thread library package ULTRA which supports the fixed-priority preemptive scheduling and cyclic executive scheduling as well as the well-known priority inheritance and ceiling protocols. The library additionally offers functions for thread management such as thread creation, deletion and synchronization (mutex locks and IPCs). We summarize several key functions of the library in Table 4.

Library call	Important Param.	Description
thread_init()	User scheduler	Initialize the thread system
thread_create()	Thread priority	Create a thread
thread_exit()	Exit status	Terminate a current thread
thread_prioctrl()	Priority	Change priority
thread_sleep()	Number of Ticks	Sleep for the ticks
thread_join()	Thread ID	Wait for a specified thread to finish
mbox_create()	Mailbox ID	Create a mailbox
mbox_send()	Mailbox ID	Send message to a mailbox
mbox_receive()	Mailbox ID	Retrieve a message from a mailbox
mutex_proto()	Protocol type	Set a protocol type
mutex_create()	Ceiling priority	Create a mutex variable
mutex_lock()	Mutex ID	Acquire a mutex lock
mutex_unlock()	Mutex ID	Release a mutex lock

Table 1: ULTRA Thread library functions.

5 Experimental Results

We performed extensive experiments with the ARX kernel in order to measure its performance and validate the correctness of its implementation. In doing so, we selected three representative applications, one of which was a hard real-time application, and the others were soft real-time applications. Specifically, they were a control program for computerized numerical control (CNC) machines developed by Kim *et al.* in [9], the Berkeley MPEG video decoder [14], and the Dhrystone Benchmark version 2.1 [20]. We ported these programs on the ARX kernel. We did not work with general time-sharing applications, since they cannot affect real-time processes.

Our experimental machine was a PC with an Intel 133 MHz Pentium processor and 32M bytes of main memory. We made our measurements using a 64-bit cycle counter implemented in the Intel Pentium processor [11]. Being initialized at the bring-up time, this cycle counter increments every machine cycle. In our experimental machine, 133,000 cycles equals to $1ms$. We set $2ms$ to the CPU time quantum and $1ms$ to the interval timer. During the experiments, our objectives were to (1) carefully measure the upcall overhead; (2) observe how the responses of soft real-time processes varied with allocated shares; and (3) check if hard real-time guarantees could be made.

5.1 Upcall Performance

An interrupt handler in the ARX kernel calls `upcall()` function, before it returns from the interrupt. Being invoked, the function prepares an event, enqueues it to an event queue, and transfers control to the user-level scheduler. Since the `upcall()` function is used uniquely in our kernel, its

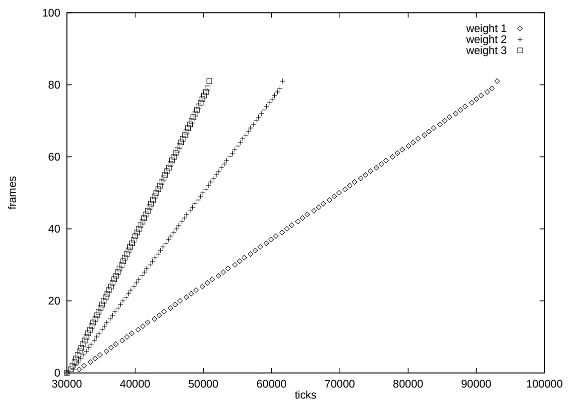


Figure 5: The number of frames decoded by three Berkeley MPEG video decoders with their weights being 1, 2, and 3, respectively.

execution incurs an extra overhead to the system. For the experiments, we measured the execution times of 20,000 invocations of `upcall()`. The following table summarizes the result.

Min	Max	Average
2.68μs	13.43μs	4.1μs

The first two columns in the table are the minimum and maximum upcall times. The last one gives the average upcall time.

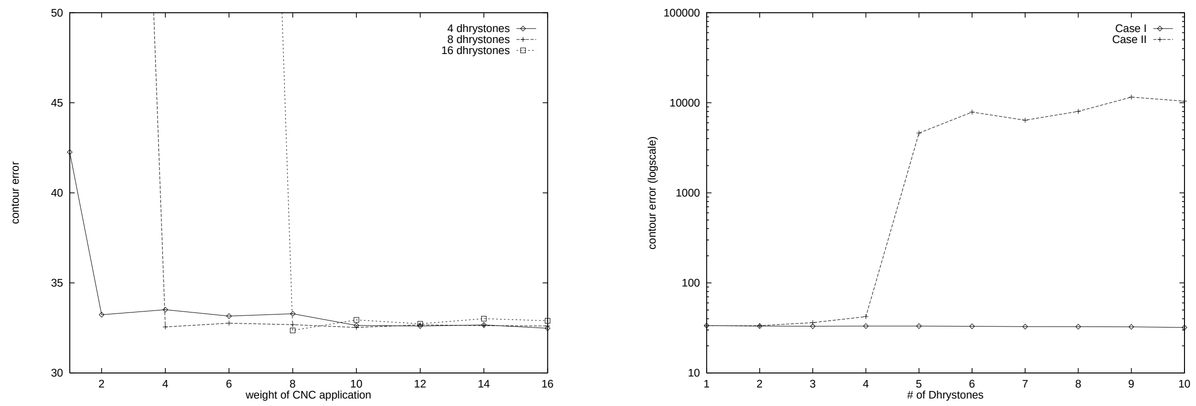
From the table, we see that there is a big discrepancy between the maximum and minimum upcall times. We found out that such variance resulted from the cold start cache misses. Since the `upcall()` function had to copy many data to build upcall arguments, cache miss penalties seriously affected the upcall performance.

5.2 Validating the Kernel Scheduler

We performed experiments with a soft real-time application to check if the WFQ algorithm was correctly implemented in our kernel. We first forked three MPEG video decoder processes giving them three different weights of 1, 2, and 3, respectively. We measured the number of frames decoded by the three MPEG processes. Figure 5 shows the results of the experiments. We can see that the number of decoded frames is proportional to their weights. This is consistent with the analytical results of the original WFQ algorithm found in [4, 13].

5.3 Hard Real-Time Performance

We experimented with the CNC controller program to show that the ARX kernel could provide hard real-time guarantees. The CNC software was implemented using two PCs, one running the controller software, and the other simulating a plant. The plant simulator was implemented as



(A) (B)

Figure 6: Contour error of the CNC controller: (A) when each Dhrystone process receives a unit weight; and (B) when the CNC controller receives a fixed 33% share of the CPU capacity (Case I), and when the CNC controller receives a fixed weight (Case II).

an infinite loop and modeled the servopacks and servo motors. It ran once every $67\mu s$ simulating ellipse cutting using the inputs from the controller software. The controller software ran on top of the ARX kernel, and communicated with the plant simulator via an I/O board. The controller software periodically sampled the cutter position, calculated the next cutter position, and sent it to the plant simulator every $2ms$. The CNC controller was implemented with seven threads. These threads were managed via the thread library package which implemented a cyclic executive. The size of a minor cycle was chosen to be $2ms$. Instead of counting deadline violations, we measured the performance of the controller software. The performance metric we used was the contour error, which is defined to be the distance between a cutter location and its corresponding reference point.

We ran a CNC controller process with several Dhrystone benchmark processes varying the weight of the CNC process and the number of Dhrystone processes. We performed two types of experiments. In the first experiment, each of the Dhrystone processes received a constant weight 1, and their numbers were 4, 8 and 16. The CNC process received variable weights which ranged from 1 to 16. We repeated the same experiment ten times and took the average contour error. Figure 6 (A) shows the results of this experiment. The X-axis shows the weight of the CNC process, and Y-axis the contour error. Observe that the CNC process yielded a small contour error which was below 35 units, when it ran with 4 Dhrystone processes and its weight was 2. We can see that in this particular case, the CNC process got one third of the total CPU capacity, or a relative weight of $\frac{1}{3}$. We can further observe that the CNC process needed more weights for an acceptable performance, as the system load increased with the number of Dhrystone processes. It is very interesting to see that the CNC process demonstrated a decent performance, when it was given about 33% of the CPU capacity, regardless of the system load.

To further verify this observation, we performed another experiment. We also ran a CNC process varying the number of Dhrystone processes. However, we arranged the CNC process to get 33% of the CPU capacity all the times and others to get the rest. Figure 6 (B) shows the result of this experiment. The graph with a solid line denotes the case where the CNC process received a fixed share (Case I). It clearly shows that the CNC performance was not affected by the number of other Dhrystone processes. On the other hand, the graph with a dotted line denotes that case where the CNC process received a fixed absolute weight (Case II). In that case, the CNC process was seriously affected by Dhrystone processes, since the share of the CNC process decreased with the number of Dhrystone processes.

6 Conclusions and Future Work

In this paper, we have presented the ARX kernel which is capable of supporting user-level threads for real-time applications. The design goals of the kernel were to provide (1) flexible and predictable scheduling services, (2) effective management of kernel-level thread blocking, and (3) efficient handling of scheduling events including timer interrupts. To achieve these goals, we proposed *dynamic stack binding*, and *scheduling event upcalls*. With dynamic stack binding, the kernel can dynamically create kernel stacks and bind them to threads. With scheduling event upcalls, the kernel can asynchronously invoke user-level schedulers in response to scheduling events. We also devised a novel, lock-free implementation of a user-level scheduler in order to reduce the upcall overhead. We have implemented a preemptive real-time kernel and a user-level thread package which supports fixed priority scheduling and cyclic executive scheduling. Our experimental results show that the kernel incurs very low overhead and offers acceptable performance with hard real-time guarantees.

During the implementation of the ARX kernel, we have found many interesting problems which remain to be solved. The first one has to do with bandwidth allocation. When we develop a real-time application on our kernel, we first have to determine the CPU capacity which can guarantee the real-time requirements of the application. In our experiments, we resorted to an engineering approach based on trial and error, for bandwidth allocation. Thus, it will be beneficial to develop a systematic method which can automatically translate hard or soft real-time requirements into a bandwidth requirement, given the performance specifications of the hardware platform. We have already conducted similar research for real-time control systems [16]. We are currently applying our preliminary results to this bandwidth allocation problem.

We are also looking to extend our kernel for multiprocessor systems. We expect that it will be straightforward to extend the event upcall mechanism such that it can handle the concurrent executions of user-level schedulers. Finally, we are working hard to demonstrate the utility of our kernel in real-world computing systems. We have already successfully ported it on an open CNC system. We currently plan to use the kernel in an intelligent automobile identification system. The results looks very promising.

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 95–109, October 1991.

[2] T. Baker and A. Shaw. The cyclic executive model and Ada. *The Journal of Real-Time Systems*, 1(1):7–25, September 1989.

[3] H. Custer. *Inside Windows NT*. Microsoft Press, Redmond, Washington, 1993.

[4] A. Dermers, S. Keshav, and S. Shenkar. Analysis and simulation of a fair queueing algorithm. *Journal of Internetworking Research & Experience*, pages 3–12, October 1990.

[5] Institute for Electrical and Electronic Engineers. POSIX P1003.4a, threads extension for portable operating systems, 1994.

[6] R. Govindan and D.P. Anderson. Scheduling and IPC mechanisms for continuous media. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, 1991.

[7] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, October 1996.

[8] P. Goyal, H.M. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. In *Proceedings of ACM SIGCOMM'96*, August 1996.

[9] N. Kim, M. Ryu, S. Hong, M. Saksena, C. Choi, and H. Shin. Visual assessment of a real-time system design: A case study on a CNC controller. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 300–310, December 1996.

[10] C. Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *The Journal of Real-Time Systems*, 4(1):37–53, March 1992.

[11] T. Mathisen. Pentium secrets. *Byte*, pages 191–192, July 1994.

[12] F. Mueller. A library implementation of POSIX threads under UNIX. In *1993 Winder USENIX*, pages 29–41, January 1993.

[13] A.K. Parekh and R.G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, December 1993.

[14] K. Patel, B.C. Smith, and L.A. Rowe. Performance of a software MPEG video decoder. In *ACM Multimedia*, August 1993.

[15] M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS multi-thread architecture. In *Proceedings of 1991 USENIX Winter conference*, pages 65–79, 1991.

[16] M. Ryu, S. Hong, and M. Saksena. Streamlining real-time controller design: From performance specifications to end-to-end timing constraints. In *Proceedings of Real-Time Applications and Technology Symposium*. IEEE Computer Society Press, June 1997. To appear.

[17] D. Stein and D. Shah. Implementing lightweight threads. In *Proceedings of 1992 USENIX Summer conference*, pages 1–9, 1992.

[18] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 288–299, December 1996.

[19] A. Tevanian, R. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young. Mach threads and the UNIX kernel: The battle for control. In *USENIX Technical Conference*, pages 185–197, June 1997.

[20] R. P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, 27(10), October 1984.