An Operating System for Real-Time Applications with Java on TINI

Qinghua Lu*, Shanshan Li, Weishan Zhang

College of Computer and Communication Engineering
China University of Petroleum, Qingdao, Shandong, 266555, China.

Abstract - **Safety is the main concern for RTOSes. Choosing a good programming language can significantly improve the safety of the RTOSes. In this paper, we investigate the advantages and disadvantages of developing RTOSes in Java. We design an RTOS named JAOS that schedules processes on a micro-controller called TINI. In the context of achieving real-time performance, we look at the issues of implementing our system design in Java. We introduce how we used Java constructs to implement the design of JAOS, and how we solved the low-level issues. We measure the performance of JAOS to investigate timing problems and provide the right information at minimal cost in performance. As a safe language, Java is suitable for coding a safety-critical RTOS.**

Keywords - Safety; Real-time; Java; JAOS; TINI

I. INTRODUCTION

Real-time applications (e.g. embeded engine controlling system in modern cars) are required to execute correctly within strict time deadlines and has much more stringent requirements than personal computer software [1]. The safety requirement of a real-time application places huge demands on the operating system that supports it.

Some real-time applications try to avoid this problem by not having an operating system. However, the application programmer has to write all the code. One advantage of using an operating system is that the programmer is better able to focus on programming the real-time task because many of the low-level details are abstracted away by the operating system. Besides, the task can be decomposed into several interacting processes. As each process is small relative to the task, the complexity of the code is reduced and its correctness increased. However, the programmer has to be able to rely on the operating system to execute every process reliably and in time. Also, the operating system must provide the low-level services the programmer requires to implement the task.

A safe programming language can significantly improve the safety of the real-time operating systems (RTOSes). Many research projects have been working on developing an operating system in a safe language to avoid crashing. The Burroughs B5000 does not have a memory management unit and relies on the Algol compiler to detect dangerous code [2]. XO/2 [3] is an RTOS developed at ETH in Zurich in an object-oriented language Oberon [4] to run on PowerPC embedded processors. A more recent project is the development of the Singularity operating system by Microsoft Research [2]. It is programmed in Sing#, a safe language based on C#. All processes run in a single virtual-address space, which is efficient because it eliminates kernel traps to perform context switches.

In this paper, we develop an RTOS named JAOS in Java. JAOS is designed to be a time-sharing system, where cooperative multiprocessing is used to schedule real-time processes. Processes are switched on a timer interrupt. JAOS executes tasks on a microcontroller named TINI [5], which is to be mounted on our flying robot to do all the fast real-time processing. The TINI platform provides an extensible Java runtime environment. We measure the flow of control, performance and reliability of JAOS.

Section 2 introduces the related work. Section 3 describes the design of JAOS and presents the code design of JAOS. Section 4 discusses the performance of TINI and JAOS. Section 5 concludes the findings of this research and outlines the possible future work.

II. RELATED WORK

XO/2 [6] is a hard-real-time system developed at ETH in Zurich to run on PowerPC embedded processors. Oberon-2 is the programming language chosen for XO/2 since it is statically and dynamically safe [7]. The design architecture of XO/2 is time-triggered. The CPU time and system resources are pre-allocated, which can lead to a waste of the system time and resources. One of the design principles is the separation of concerns [8]. The XO/2 system has been used for many research projects and commercial products. Brega argues the XO/2 system has successfully implemented the software techniques addressing safety on a system-wide level [3]. Brega points out that the Java programming language has the same level of safety as Oberon-2. This is one of the motivations for us to develop an RTOS in Java.

The JX operating system is a single address space operating system mainly written in Java [9]. The features of Java raise the level of abstraction and help to develop more robust systems in less time. Protection in JX [10]is based on the type-safety of the Java byte code instruction set.

Therefore, there is no memory space switch caused by inter-process communication and system calls. To expand the address space, MMU support can be added. Typical Java security problems, such as native methods, execution of code of different trustworthiness in the same thread, and a huge trusted class library, are avoided by JX. The code written in C and assembler is kept to minimal, which makes the system simple and robust.

Singularity is an operating system developed by Microsoft Research [2]. It uses advances in programming languages to develop an operating system that an errant process cannot crash [11]. Singularity is programmed in Sing#, a new type-safe language based on C#. All processes run in a single virtual-address space, which is very efficient because it eliminates kernel traps to perform context switches. The exclusion between processes is complete (without using an MMU for protection) with each process having its own code, data structures, runtime, libraries and garbage collector.

III. DESIGN OF JAOS

A. Overall Architecture of JAOS

Fig. 1 illustrates the overall architecture of JAOS. The Operating system (OS) is completely isolated from user applications. Splitting the responsibility in this way results in the application programmer being able to focus on the design and programming of the set of processes required to solve the application problem. The OS part provides the main components of JAOS, including OS Methods (Table 1), OS Processes (Table 2), OS Tables (Table 3) and Supervisor Calls (Table 4). The operating system is an instance of an OS class. The JVM, JVM runtime, and hardware are considered to be the machine. JAOS runs as a process on top of the JVM runtime.

As shown in Table 1, a few method work together to provide the run-time kernel of the OS. Much of the work of the OS and all the work is done by applications. Therefore, the task of the OS kernel is to schedule processes. The Main method is called to start the OS by enabling timer interrupts and then calling the scheduler to schedule the first process. Performance probes are placed in the scheduler to measure process performance.

The OS processes (Table 2) do the work of the OS apart from scheduling. The timer process, which scheduled in response to the timer interrupt, sets flags to tell the scheduler when to run time-triggered processes. The other processes in Table 2 handle common OS functionality. Note, the event monitor process is an application process not an OS process because the events are specified to each application.

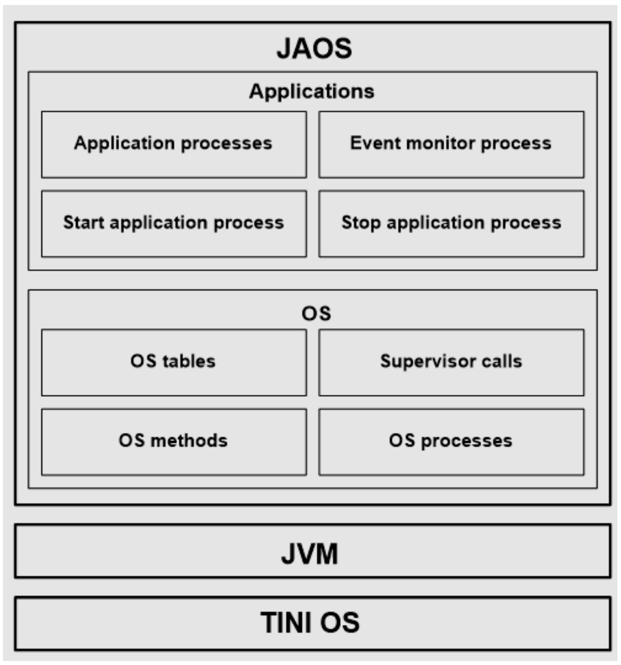

Fig. (1). Overall architecture of JAOS.

The scheduling of processes and other OS functionality requires a number of tables. These are given in Table 3.

To request work by the OS, processes execute supervisor calls (Table 4). The calls allow a process to start and stop other processes, to communicate with other processes, and to respond to events. By restricting this functionality to supervisor calls, we stop poorly written application code corrupting the OS table.

TABLE 1. OS METHODS

Name	**Description**
Main	Initializes OS tables and processes, enables timer interrupt, enables processes and calls scheduler to start OS
Scheduler	Decides which process is to run and dispatches it
Enable timer interrupt	Enables/disables timer interrupt
Timer interrupt handler	Sets flag to run timer process and handles time out
Performance probe	Collects performance data and places it onto a circular buffer
Process	A method of a process object that performs computation

TABLE 2. OS PROCESSES

Name	**Description**
Timer process	Maintains the timer table and sets flags for processes to run
Message Monitor process	Checks for the arrival of messages
Performance Analysis process	Analyses data collected by performance probes
Timeout Report process	Reports on the timeout of a process
Garbage Collector process	Runs when time available to clean up heap
Idle process	Runs when no other process requires the CPU, enables the event monitor to run
Terminate process	Disables timer interrupt and resets tables to stop scheduler

TABLE 3. OS TABLES

Name	**Description**
OS table	For OS variables
Process table	Dynamic part of process control block (process state)
Scheduler table	For scheduler variables
Memory table	For memory variables
Event table	For event variables
Message table	For message variables
Performance/testing data table	For performance and testing data
Circular Buffer table	For separation of real-time concerns, and for performance analysis
Common Data table	For common data

TABLE 4. OS SUPERVISOR CALLS

Name	**Description**
Run Process	Sets the execute flag in the scheduler table for a process that has been loaded and enabled. Scheduler runs process
Stop Process	Resets execute flag in the scheduler table
Get Message	Gets a message object – array of ints, floats or string
Send Message	Writes message into message buffer and sets available flag
Receive Message	If there will get message and reset available flag, else will return and set process up to waits for the message, giving up processor
Release Message	Returns message resource to OS
Get circular list	Creates a circular list object
Add to Circular List	Adds values to a circular list
Remove from Circular List	Removes values from a circular list
Write common data value	Writes values to common data area
Read common data value	Reads values from common data area
Wait Event	Waits for an event
Wait	Goes into wait state for n clock ticks – a form of self scheduling
Load Process	Loads a process - set up OS tables using values in process control block
Remove Process	If process is running stops it and clears out OS tables
Enable Process	Add a process to Scheduler table
Disable Process	Removes a process from Scheduler table
Change priority	Changes the priority of user processes by moving them in process table
Simulate event	Switch from hardware event to software event simulator for testing
Get OS Tables	Copy the current value of all OS tables for use in debugging, testing and performance measurement
Library of event handlers	A library of event handlers.

TABLE 5. PROCESSES IN A TYPICAL USER APPLICATION IN PRIORITY ORDER

Name	**Description**
Event Monitor process	Polls for I/O event
Application specific process	Application specific code
Start Application process	Sets up processes to get the application to be run by the OS
Stop Application process	Stops all processes in the current application

The purpose of the OS is to run applications An application consists of several communicating processes. All applications are started by a Start Application process where responsibility is to set up the processes required to perform the application and schedule at least one to run. The purpose of the Stop Application process is to gracefully shut an application down. The event monitor process polls I/O to check for extend events and sets scheduler flags to start application processes to respond to the events.

Getting the system running is the responsibility of the Main method (Algorithm 1), which starts the operating system, enables the Start Application process, and calls the scheduler loop. The scheduler runs the Start Application process, which starts the processes to perform the user application.

Algorithm 1 Main method

```
Declares an object of type OS – calls the OS constructor, which
Declares instances of all tables
Initialises all table values to zero
Initialises the OS table
Reads current time and sets to previous time to correctly initialise the timer
Loads and enables the following processes, which are considered to be part of the OS: Timer, Garbage Collector, Timeout Report, Performance Analysis and Idle.
Loads and enables the Start Application Process: to load, enable and run application processes.
Sets the Start Application Process and the Idle process to run
Enables timer interrupts
Calls the scheduler method, which only returns to Main when terminate process is called.  Then Main should exit gracefully.
```

Fig. 2 shows the timing control flow of JAOS. The JAOS system schedules processes on a timer interrupt, which we have simulated in the Idle process for much of our testing. An interrupt handler sets the execute flag of Timer process in Scheduler table, and executes the timeout function if any process in the Scheduler table has gone over time.

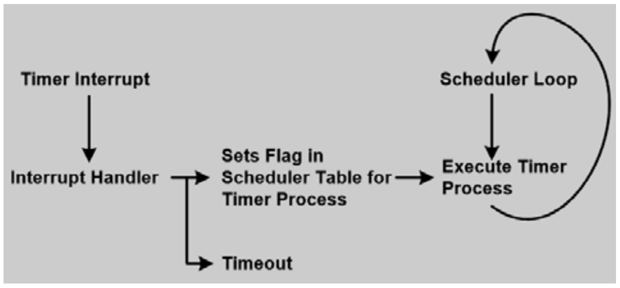

Fig. (2). Timing flow control in JAOS.

B. Scheduler

The scheduler runs at the completion of each process and should run at least every clock tick. It is responsible for giving the CPU to the processes that want it, in priority order. The scheduler loop only exits when a call to the terminate process resets all execute flags in scheduler table. The scheduler checks the flag of each process in the scheduler table. If the process is ready to run, the scheduler resets its execute flag to false in the scheduler table. The scheduler sets timeout counter and current process number in OS table. Then the scheduler starts the process and pass state to it. The process executes and returns to the scheduler. If the performance testing flag is set, the scheduler calls performance probe method before and after calling the process. When a process is timed out it is set to return to the scheduler as if it was a normal exit so that the scheduler does not have to clean the timed out process. When a process goes into wait, it must call a method to set up the timer table values, then it must set its state values, then it returns to the scheduler. All processes must execute quickly and return to the scheduler. The scheduler is held in an infinite loop by the idle process in the scheduler table always being enabled to run. A call to the terminate process will reset all enable flags in the scheduler table and the scheduler will return to the Main method, whose task it is to exit gracefully.

Algorithm 2 Scheduler

```
$i = 0$
WHILE there is a process to run
$i = i + 1$
IF process $i$ is ready to run
Reset process execute flag in scheduler table
```

```
Set timeout counter in OS table
Set current process number in OS table
IF performance testing flag set in OS table THEN
Call probe method
Call process
ENDIF
IF performance testing flag set in OS table THEN
Call probe method
$i$ = 0
Set current process number in OS table
ENDIF
ENDWHILE
```

C. Implementation

The majority of JAOS is written in Java. Java can implement all high-level functions in the design. The OS tables are stored as Java arrays and accessed by the methods of the relevant table class. Each process is written as a subclass of the Process class by using the inheritance of Java. Each OS process is constructed in the Main method by declaring an instance of its class. Each user process is constructed in the process method of the StartApplication class. schedulerInfiniteLoop() method is a method of the Scheduler class. schedulerInfiniteLoop() calls the OS processes on the first loop, then the StartApplication process enables each user process to run. The scheduler calls processes using Java Reflection.

The timer interrupt handler should be connected to the hardware interrupt of TINI. In initial testing of JAOS, the timer interrupt is simulated in the Idle process. The interrupt updates the clock value every tick.

When an interrupt occurs, the processor stops the thread of execution of the current process at the end of the current instruction, saves some system state and vectors to an interrupt handling function called timerInterruptHandler(). When the interrupt handler completes servicing the interrupt it normally returns to the hardware, which restores the state and continues the thread of execution of the current process. In order to implement some operations in response to interrupts (for example a time out), interrupt handlers may have to change the return address of the process that it interrupts so that JAOS can take the processor away from that process.

As this type of operation is potentially dangerous and can cause failure to meet deadlines, the only time an interrupt handler is allowed to return to a different address is in a time out. When a time out occurs, the interrupt returns to the exit address of the interrupted process so that it returns to the scheduler in the normal way. These low-level functions in the timerInterruptHandler() method cannot be written in Java. TINI provides TNI (TINI Native Interface) for programmers to call native code in Java code. Therefore, we implement low-level functions in assembly language supported in TNI.

In TINI Native API, there is a function named System_SaveJavaThreadState used to save the Java state for the current thread. We have a native method called Native_SaveState() to call this function. When an interrupt occurs, the Native_SaveState() method will be called by the timerInterruptHandler() method to save the state of current running process.

There is a function named System_RestoreJavaThreadState used to restore the Java state for the current thread. We have a native method called Native_RestoreState() to call this function. When the interrupt handler completes servicing the interrupt, the Native_RestoreState() method will be called by the timerInterruptHandler() method to restore the state and continue the thread of execution of the current process.

The timer interrupt is programmed as a single background thread by using the java.util.Timer and java.util.TimerTask classes. The timer interrupt is scheduled as a TimerTask object for repeated execution at regular intervals by a Timer.

IV. PERFORMANCE MEASUREMENT

A. Testing Clock Simulation, Timer Interrupt Handler and Process Overhead

We measured the time that clock simulation takes and the execution time of the timer interrupt handler. It takes 23msec to simulate a clock tick while it takes 13msec to execute the timer interrupt handler.

Process time consists of reading the clock time, code execution time and process overhead time. The read clock calls are to obtain the time data to measure the performance. Thus, they represent the time overhead of performance probes. We measured the overhead time of calling a process. The process overhead time is 3msec.

B. Testing the Flow of Control of JAOS

We validate that the code achieves our system design by using the performance data. Three synchronous processes are required to run to maintain the JAOS running. These are Timer process, Idle process and Test process. Test process is an application process on JAOS, which we wrote for performance measurement. After running test process a certain number of times, the scheduler runs the Performance Analysis process, which prints out the measured data.

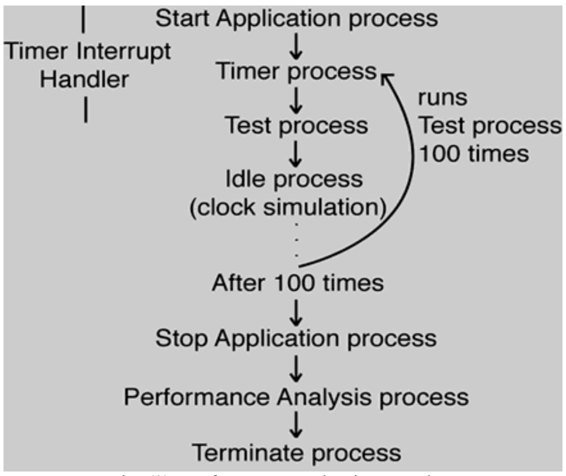
Timer Interrupt Handler
Start Application process
Timer process
Test process
Idle process (clock simulation)
runs Test process 100 times
After 100 times
Stop Application process
Performance Analysis process
Terminate process

Fig. (3). Performance evaluation template

Fig. 3 shows a template of the test system, which can be used to measure the performance of any application simply by running the application processes as the Test process. The Test process, which is set to run by Start Application process, runs every 2 clock ticks. We wrote different test processes to test the flow of control of JAOS. One of the test processes is a process that turns on or turns off a LED on the TINI. The clock is simulated in the Idle process, which is set to tick every 200msec. After 100 executions of the scheduler loop, the Stop Application process stops the Test process. In the scheduler loop, performance probes are called before and after the execution of each process to collect performance data. The performance probes put process number and current time onto a circular buffer. Then the scheduler runs Performance Analysis process to produce the performance trace, and the Terminate process to terminate the JAOS system.

TABLE 6. PART OF TEST DATA COLLECTED BY PERFORMANCE ANALYSIS PROCESS

Process no.	Process name	Time/msec	Process time/msec	Scheduling time/msec
1 start	Timer process	5600740		
1 finish	Timer process	5600860	120	
6	Test process	5600880		20
6	Test process	5600900	20	
31	Idle process	5600930		30
31	Idle process	5600970	40	
31	Idle process	5601000		30
31	Idle process	5601040	40	
1	Timer process	5601060		20
1	Timer process	5601180	120	
31	Idle process	5601210		30
31	Idle process	5601250	40	
31	Idle process	5601280		30
31	Idle process	5601330	50	
1	Timer process	5601340		10
1	Timer process	5601460	120	
6	Test process	5601480		20
6	Test process	5601510	30	
31	Idle process	5601540		30
31	Idle process	5601570	30	
31	Idle process	5601600		30
31	Idle process	5601650	50	

A scheduler loop consists of scheduling time and the process time of a process. Scheduling time is the time that the scheduler takes to select which process is to run. Process time is the execution time of a process. The execution time of the start performance probe is included in the scheduling time, and the execution time of the finish performance probe is included in the process time. Previously we measured the probe time to be 7msec. In order to maintain consistency of data, we have left this overhead in the calculations in Table 6 and 7. Table 6 shows the data that we collected.

TABLE 7. RESULT OF TESTING THE FLOW OF CONTROL OF JAOS

Process name	Process time/msec (medium ± max)	Scheduling time/msec (medium ± max)
Timer process	120±10	10±10
Test process	30±10	20±10
Idle process	40±20	30±10

Table 7 lists the average process time and average schedule time for each process. These times are all constant to the precision of our time measurement. We found that a process always has the same scheduling time, and that the process execution time of a process is fairly consistent. We also note that scheduling time varies with process number, because the scheduler iterates down the scheduler table until it finds a process to run. Consequently, scheduling time increases with the process number.

Fig. 4 shows the flow of control of JAOS with the execution time of each process. We observe that performance data that we collected conforms to the design of JAOS.

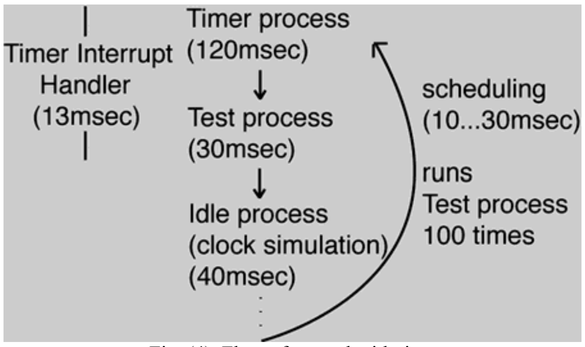
Timer Interrupt Handler (13msec)
Timer process (120msec)
Test process (30msec)
Idle process (clock simulation) (40msec)
scheduling (10...30msec)
runs Test process 100 times
Fig. (4). Flow of control with time

Fig. 5 shows the time relationships of the processes running on JAOS in the test. The Timer process (p1) runs every clock tick, the Test process (p6) runs every 2 clock ticks, the Idle process (p31) runs the rest of time. Table 6 shows that the Idle process runs twice every clock tick. Also, when there are no processes to run, JAOS runs the Idle process.

p1: Time = 1, Phase = 1
p2: Time = 2, Phase = 2
idle: rest of time – OS is checking for events etc.

Time means the number of clock ticks between executions. Phase is the clock tick relative to first.

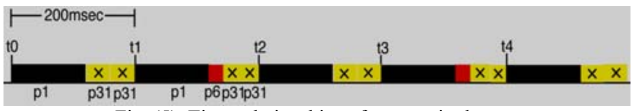

Fig. (5). Time relationships of process in the test

C. How Long should the Clock Tick be?

A significant design parameter in JAOS is the time between clock ticks, whether the clock is simulated or a hardware interrupt. We chose a target, based on the 20/80 rule, that JAOS spends 20% of time performing OS tasks including running the Timer process, leaving 80% of time for applications.

We have obtained that the timer interrupt simulation and the timer interrupt handler take 23msec, process scheduling takes 20msec, and the Timer process and one performance probe take 120msec. On these basis, the time between clock ticks should be 637msec for application to run.

When using the simulated clock tick, the Idle process has to run to simulate the clock. To get a regular tick, we should allow sufficient application time for an integral number of ticks.

Number of ticks=application time/Idle process time=637/40=16

Therefore, the duration is sufficient to allow multiple checking for clock tick even when running a number of processes.

D. Reliability Testing of JAOS

We evaluate the reliability of JAOS working for a long time. To maintain the JAOS running, four synchronous processes are required to run, which are Timer process, Idle process, Test process, and Workload process. Test process and Workload process are set to run every 2 clock ticks. A simple Test process is a process that turns on or turns off the LED on the TINI. The Workload process counts the executions of itself, not them of the Test process. The Test process simply provides a workload to exercise OS functions and is not modified. To count the execution of the Test process it would have to be modified.

In Workload process, counters and execution parameters of the Workload process are maintained. These counters include the number of execution of the Workload process. The parameters include the average, minimum and maximum times between executions of the Test process. If JAOS runs for the full test time, the scheduler will set the Stop Application process, the Workload Analysis process and the Terminate process to run. The Workload Analysis process is a process that prints out the run time, execute count and timing parameters.

We previously suggested that the clock tick should be 800msec. However, we ran a test with a 200msec clock tick to learn whether the reliability test would confirm that there is a problem. We set the Test process and Workload process to run every 2 clock ticks or 400msec.We set the test time to 24 hours. That is 86,400,000msec. After running 24hours, we obtained the test result of the reliability test. The measured runtime is 86,400,070msec, and the execution count is 157,596. The average time between executions of the Workload process is 548msec, which is 37% longer than the expected 400msec.

As shown in Table 9, the total execution time is 265msec per clock tick with a 200msec clock tick or a 32.5% time overload. Therefore, we show that the 37% longer execution time is due to overload. If we set the clock tick to 400msec or 800msec, the total time per clock tick is less than the clock tick time.

Reliability testing showed a problem and analysis showed the cause. The good news is that overloading JAOS did not cause it to crash, the executive overload simply caused it to run late, i.e. it slowed down gracefully even though it failed to meet the deadlines.

TABLE 9. ANALYSIS OF THE PROBLEM IN THE RELIABILITY TESTING

		200msed/tick	400msec/tick	800msec/tick
Timer process	Scheduling time	10msec	10msec	10msec
	Timer process	120msec	120msec	120msec
Simulated timer interrupt	Scheduling time	30msec	30msec	30msec
	Idle process	40msec	40msec	40msec
Test process	Scheduling time	10msec (20/2)	20msec	40msec (20*2)
	Test process	15msec (30/2)	30msec	60msec (30*2)
Workload process	Scheduling time	10msec (20/2)	20msec	40msec (20*2)
	Workload process	30msec (60/2)	60msec	120msec (60*2)
Total Time		265msec (>200msec)	310msec (<400msec)	460msec (<<800msec)

V. CONCLUSION

We developed an operating system for real-time applicatios in a safe language, Java. The design of JAOS is a time-sharing design switching tasks on a timer interrupt. Each application task is decomposed into several interacting processes. As each process is small relative to the task, the complexity of the code is reduced and its reliability is increased.

In JAOS, application code is separated from the OS code. The programmer only needs to write application code and make no changes to the OS. Thus, they can focus on programming the real-time task because many of the low-level details are abstracted away by the OS.

As a safe language, Java is suitable for coding a safety-critical RTOS. The Java compiler handles potentially unsafe operations rather than the programmer. Also, Java includes run-time support to catch and handle run-time errors. However, the low-level operations cannot be coded in Java, which is a main problem of writing an RTOS in Java. We can only rely on the native interface provided by the JVM. Therefore, the relevant documentation should be sufficient for the developers to study.

We investigated the timing problems by a set of performance measurements. The performance data we collected conforms to the design of JAOS.

One surprise was how poor the performance of the TINI JVM is. The developers of TINI claim that each thread is assigned an 8 msec time slice on TINI [5]. The test data shows that it takes 0.063 msec to execute a fundamental instruction unit on TINI. That is 126 fundamental instructions per slice, which means it does not do much in any slice. Therefore, the time slice used in TINI OS should be longer. Also, the performance data of TINI shows that the speed of TINI is much slower than we expected. The documentation of TINI is poor, which is inconvenient for doing research on the TINI board. Also, there are some omissions in the TINI API, such as reflection [5]. These hampered the development of JAOS. Therefore, we conclude that TINI is better suited to developing stand alone programs than to developing an RTOS.

CONFLICT OF INTEREST

We confirm that this article content has no conflicts of interest.

ACKNOWLEDGMENT

This project is supported by National Natural Science Foundation of China (Grant No. 61402533), “the Fundamental Research Funds for the Central Universities” (No. 14CX02140A) and “the Scientific Research Foundation of China University of Petroleum” (No. Y1307021).

REFERENCES

[1] Laplante, P.A., Real-Time Systems Design and Analysis, 3rd Edition, Wiley-IEEE Press, 2004.

[2] Tanenbaum A.S., Herder, J.N. and Bos, H., “Can we make operating systems reliable and secure,” IEEE Computer, May 2006, pp 44-51.

[3] Brega, R., A Combination of System Software Techniques Aimed at Raising the Run-Time Safety of Complex Mechatronic Applications, PhD thesis, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, 2002.

[4] Mössenböck, H. P. and Wirth, N., “The Programming Language Oberon-2,” Structured Programming, 12(4), pp.179-195, 1991.

[5] TINI, TINI networked microcontroller, http://www.maxim-ic.com/products/microcontrollers/tini/.

[6] Brega R., Tomatis N., Arras K.O., “The Need for Autonomy and Real-Time in Mobile Robotics: A Case Study of XO/2 and Pygmalion”, Proceedings of IEEE/RSJ Int. Conference on Intelligent Robots and Systems, Takamatsu, Japan, 2000.

[7] Tomatis N., Terrien G., Piguet R., Burnier D., Bouabdallah S., Arras K.O., Siegwart R., “Designing a Secure and Robust Mobile Interacting Robot for the Long Term,” Proceedings of IEEE International Conference on Robotics and Automation (ICRA'03), Taipei, Taiwan, 2003.

[8] Tomatis, N., Brega, R., Jensen, B., Arras, K., Moreau, B., Persson, J., Siegwart, R., “A Complex Mechatronic System: From Design to Application,” Proc. IEEE/ASME Int. Conf. on Advanced Intelligent Mechatronics (AIM), Como, Italy, July 8-11, 2001.

[9] Golm, M., Felser, M., Wawersich, C. and Kleinoeder J., A Java Operating System as the Foundation of a Secure Network Operating System, Technical Report TR-I4-02-05, August 2002.

[10] Golm, M., Felser, M., Wawersich, C. and Kleinoeder J., “the JX operating system,” in Proceedings of the USENIX Annual Technical Conference, pp. 45–58, Monterey, CA, June 2002.

[11] Hunt, G., Larus, J., Abadi, M., Aiken, M., Barham, P., Fahndrich, M., Hawblitzel, C., Hodson, O., Levi, S., Murphy, N., Steensgaard, B., Tarditi, D., Wobber, T., and Zill, B., “An Overview of the Singularity Project,” Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[12] McCluskey, G., “Using Java Reflection,” Java Developer Connection, 1998.

[13] Sosnoski, D., Java programming dynamics, Part 2: Introducing reflection, < http://www.ibm.com/developerworks/library/j-dyn0603/>