Decomposition of Preemptive Scheduling in the Go! Component-Based Operating System

Greg Law & Julie McCann,
Dept. of Computing,
City University,
London, UK
email: {gel,jam}@soi.city.ac.uk

Abstract

Embedded systems are required to exhibit ever increasing functionality while continuing to use minimal resources. The next generation of embedded operating systems must support protection with very low overheads, as well as being (dynamically) configurable. Go! is a prototype component-based system that runs natively on the Intel 386 based PC. Its novel protection mechanism means that components are (optionally) protected from one another, but exhibit very low overheads. Furthermore, components can perform system tasks previously considered bound to the kernel (such as interrupt handling and preemptive scheduling).

Go! does not provide multithreading, but is constructed so that components comprising a 'library operating system' may provide (preemptive) multithreading with relative ease. This paper describes that support, and goes on to present the scheduling provided by GTE (a 'proof-of-concept' library operating system built on top of Go!). We show that decomposing multithreading into thread components, an interrupt-dispatcher and a scheduler is practical, useful, stable, and performs well.

1 Introduction

Personal Digital Assistants (PDAs) and wearable computers currently provide a small set of applications using non-standard communication protocols and operating systems. The move toward a more comprehensive handheld device that runs as a super PDA (ie it is a phone, two-way radio, television, pager, handheld computer, pointing device etc) requires support from an operating system that is lightweight, extensible (flexible or adaptable) and that performs well. That is, functionality such as improved application-defined protection schemes, dynamic component replacement and application participation in resource management is now required from the OS. The Go! [7] component-based operating system provides a solution to these problems.

Decomposition of the operating system can result in improvements in flexibility, systems-software engineering and availability. However, conventional protection models make it difficult to decompose fundamental system services, (such as interrupt handling, scheduling and context switching) into separate components. Most current component-based operating systems employ a Micro-kernel to perform these tasks, at least at some level [5, 2]. These Micro-kernels exhibit high levels of coupling, and consquently suffer the associated draw-backs (e.g., poor software engineering, flexibility and configurability [8]).

Go! is different. Its analogue of a kernel is the *ORB* (Object Request Broker). Unlike a kernel, the ORB provides component-management only. That is, the ORB supports inter-component communication (over RPC), component instantiation and destruction, and type registration and de-registration. There is no support for many of the services usually expected of even the smallest kernels (such as Exokernels [1]). E.g., the ORB does not handle interrupts or paging, not even just to farm them out to 'user-space' — the ORB is blissfully unaware of such details. If services such as interrupt management or multithreading are required they must be implemented by user components. This way tasks that (while related) have quite separate implementations (such as component-management, memory-management, interrupt dispatching and thread preemption/scheduling) can be separated into distinct

components.

As a proof of this concept, a library operating system has been developed for Go! that provides interrupt management, preemptive multithreading and memory management. This library operating system is called GTE (Go! Test Environment). GTE's preemptive scheduler, as well as its interaction with the rest of GTE and the ORB is described in this paper.

The structure of this paper is as follows. Section 2 presents an overview Go!, and Section 3 describes Go!'s support for library OS multithreading. Section 4 describes the design and implementation of scheduling in GTE. Section 5 reports on performance experiments and their results, and section 6 concludes.

2 An Overview of Go!

Go! is a prototype, component-based operating system that runs natively on the Intel 386 based PC. The fundamental OS primitive provided by Go! is the *component*. Go! enables components to be protected from one another in a novel way, which allows Go! itself to be responsible only for component management. More accurately, 'application-level' components are able to perform system tasks such as interrupt handling because of the 'code-scanning' technique developed with Go!.

Code-scanning works as follows: all code executes in the most privileged mode on the micro-processor (Ring 0 on the Intel 80386). Untrusted components have their code-section scanned prior to installation on the system; components found to contain instructions which they are not sufficiently privileged to execute are rejected.

Memory protection is enforced through the Intel's segmentation hardware: each component instance's data reside in their own data segment, and each instance is of some type, the code of which resides in its own segment. An instance can be identified by its data segment, and a type by its code segment.

The code-scanner considers a segment-register load to be a privileged operation, so that holding a segment's selector in a segment register is a capability to access that segment [6]. In other words, a component's data can be accessed only by its code segment — encapsulation is enforced. This means that a context switch can be effected by loading the code, data, and stack segment registers with new values (a segment-register load takes just 3 cycles on a Pentium). This reduces local null-RPC latency to around one order of magnitude lower than high performance OSs like L4 [4]. Note that an inter-segment branch is considered a segment-register load, and so prohibited. The exception is calls into the ORB's (well-known) code segment at a few (well-know) offsets (e.g. `call 8:16` issues a (local) RPC to another component).

Code-scanning obviously prevents self-modifying code[1] and embedding arbitrary data in code segments. However, both techniques are employed rarely and modern micro-processors impose significant performance penalties on their use. In the rare event that components do require such functionality, those few components can execute while the processor is in user mode.

Implementing code-scanning on a CISC machine is non-trivial. The variable-length instructions mean that it is difficult (although rarely impossible) to determine what bytes will be executed as code and what bytes will be read as an instruction's immediate data. The presence of indirect branches (including sub-routine returns) compounds this problem. In those few cases where it is not possible to determine that it is safe to execute a component in kernel mode, that component can be executed in user mode (and incur the associated performance penalties).

As well as improved performance, this model allows the operating system to be truly decomposed: a normal 'application-level' component can implement system behaviour (such as interrupt handling) [2]. Requiring a segment-register load to perform a context switch, along with the single processor mode also drastically reduces inter-component communication times. This allows improved systems-software engineering through finely-grained, protected components without sacrificing performance (thus requiring less resources).

Although the ORB does not provide many 'system services', it is designed so that library operating systems can be easily constructed from components. That is, although the ORB does not provide multithreading, interrupt-handling, paging or memory-management, the implementation of these services was considered during Go!'s design.

For a more in depth discussion of the Go! protection model, see [7].

[1] Note that self-modifying code does not include 'dynamic code generation', such as with a Java JIT compiler — a JIT does not modify *itself*, but produces new code, which would be fed through the code-scanner prior to execution.

[2] The code-scanner must know to trust certain system components. It is up to the library operating system to decide what components are trusted.

3 Go! Multithreading Support

Although Go! does not provide multithreading directly, it does have support for its implementation by library operating systems. In order to understand the issues involved in this support, one must be familiar with the ORB's thread-tunnelling RPC mechanism. Local RPC is the principle means of inter-component communication in Go!. The caller (or client) requests that the ORB invokes a method on another component (the callee, or server). Once the callee has completed its work it returns control to the caller, again via the ORB. This is illustrated in Figure 1.

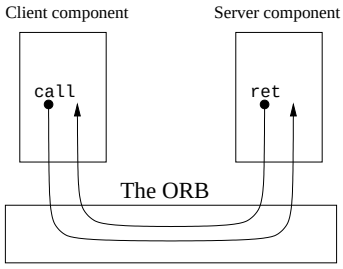

Figure 1: Inter-component RPC via the ORB

The (single) thread of control migrates between components during RPCs. The caller's stack frame is (optionally) protected from the callee component. Although no time-out mechanism is imposed, when the callee issues a 'return' request to the ORB, control is guaranteed to return to the instruction immediately following the most recent call. In effect, an inter-component call is guaranteed to return properly, but no bounds are imposed on the time this takes (the callee not returning is the same as the call taking infinite time). As with conventional systems, if the caller does not trust the callee to return, it should arrange to respond to some time-out (i.e., this is the 'halting problem' [9]).

The ORB's call and return primitives might be implemented as follows:

`call( comp target, int method )`

1. load the ORB data segment
2. validate `target` and find its data segment
3. validate method number
4. push out-going component ID onto stack
5. protect caller's stack frame by:
   (a) push the existing `old_limit`
   (b) set `old_limit` to stack segment's limit
   (c) push the existing `old_sp`
   (d) set `old_sp` to stack-pointer
   (e) set stack segment limit to stack-pointer
6. increment `call_depth` and callee's `call_count`
7. locate callee's method table
8. load callee data segment
9. place caller's ID in general-purpose register
10. jump into callee code segment at offset indicated by method table

`return()`

1. load the ORB data segment
2. unprotect the caller's stack frame by:
   (a) set the stack's limit to `old_limit`
   (b) set the stack-pointer to `old_sp`
   (c) pop `old_sp`
   (d) pop `old_limit`
3. decrement `call_depth` and callee's `call_count`
4. pop caller's ID
5. load caller's data segment
6. return to caller (using the x86 `retf` instruction)

The most important support for multithreading is that the RPC call and return primitives are reentrant. However, because the ORB has no notion of threads, blocking synchronisation methods (such as 'mutexes') are not an option. Also, because Go! does not manage interrupts (and due to performance considerations) it cannot simply acquire a large spin-lock during execution of the RPC primitives. Instead, non-blocking synchronisation [3] is used to control concurrent access to data shared between threads (e.g. components' `call_count` variable referred to above).

Even non-blocking synchronisation is not possible for variables such as `old_sp`, as these are specific to the current thread. In this case, the ORB supports the

creation of new 'stack-contexts' and the switching between them. That is, there is not just one instance of variables such as `old_sp` as implied above, but (effectively) a variable-sized array. The 'stack-context' consists of the `old_limit`, `old_sp` and `call_depth` shown above (along with several other data, the details of which are beyond the scope of this paper). Library operating systems switch between these stack-contexts when scheduling a new thread — that is, the ORB is only reentrant as long as the appropriate stacks-context is used for each thread. Note that the operation to switch between stack-contexts is not reentrant — it is up to the component invoking the switch to ensure mutual exclusion. Note also that switching stack-contexts does not alter the stack itself — this too is delegated to the caller (that is, the library operating system).

The Go! ORB defines a few 'standard types' (types which are integrally part of Go!, and do not need to be loaded explicitly: analogous to UNIX processes such as `swapper`). This includes the `stack` component type. The `stack` type has no methods, other than the compulsory *constructor* and *destructor* methods (which are implemented by the ORB for all standard types). During construction of an instance of the `stack` type, a new stack-context is created and associated with the new component. Stack-contexts are identified by their associated `stack` instance — that is, when calling on the ORB to switch to specific stack-context, the relevant `stack` is specified. It is anticipated that library operating systems will associate a `stack` component with each thread, and load the stack-segment register with the `stack`'s data segment when scheduling that thread.

4 Scheduling in GTE

The GTE library operating system provides simple, round-robin, single-priority, preemptive scheduling. As well as the ORB, four component types are involved in providing preemptive scheduling: the interrupt dispatcher (`idisp`), the scheduler (`sched`) a component to represent threads (`thread`), and the `stack` type introduced in Section 3. The IDL of these interfaces is:

```
interface comp {
   void ctor();
   void dtor();
   void catch(comp xcp, uint offs);
};

interface thread : comp {
   void ctor(comp startc, uint startm, size_t s);

   void start();
   void resume(uint gp_regs[8], uint eip);
   stack get_stack();
};

interface idisp : comp {
   void attach(thread t, uint vector);
   void unattach(uint vector);
   void intr_done(uint vector);
};

interface sched : comp {
   void intr(thread handler);
   void attach(thread t, uint flags);
   void unattach(thread t);
   void tick();

   void block(thread hand\_off = 0);
   void unblock(thread t );

   thread get_current();
};
```

To create a new thread, it is necessary to create an instance of the `thread` type and attach it to `sched`. As shown, the `thread` type's constructor takes 3 arguments: the component and method-number where the thread will commence execution (`startc` and `startm` respectively) and the size of its stack (`s`). As part of construction, `thread` will create a `stack` component of the relevant size (this `stack` can be retrieved by calling the `thread`'s `get_stack()` method). When `start` is called, the `thread` will (at least) call method `startm` on component `startc`.

In order to *activate* a thread, it must be `attach`ed to the single instance of the `sched` component. The `flags` parameter specifies scheduling information, including whether or not the thread should be attached in a blocked state and what action to take on the thread's termination. When the `thread` is first scheduled, its `start()` method is invoked. Depending on the `flags` parameter when the thread is `attach`ed, its `resume` method will be called at the beginning of each subsequent 'time-slice'.

The `idisp` component is used to schedule a `thread` immediately on the receipt of an interrupt on a given vector (interrupt vectors are not prioritised in the current version of GTE). Interrupt-driven `thread`s must also be attached to `sched`, otherwise they will not be executed on receipt of an interrupt.

The implementation of `sched` is somewhat intricate, and its interface (although not its implementation) is intertwined with `idisp`'s. On construction, a `thread` component is created, and used as the 'scheduler-thread'. `sched` creates the scheduler-thread so that it will call its `tick` method when first scheduled, and `attach`es the thread to itself in a blocked state. The 'scheduler-thread' is then attached to `idisp` on vector 32 (the timer interrupt). This means that the `tick` method is called on each timer tick. The `tick` method does nothing except to call the `idisp`'s `intr_done` method. After re-enabling interrupts on vector 32, `idisp`'s `intr_done` method will call upon `sched` to block the current thread (i.e. the scheduler-thread). Since blocking a thread causes a new one to be scheduled, this means that each clock-tick results in the currently executing thread being suspended, and a new one being scheduled. In summary, on each timer interrupt: (1) the current thread is suspended; (2) the scheduler-thread is woken; (3) the scheduler-thread is blocked; (4) the next thread is resumed. This is demonstrated in Figure 2.

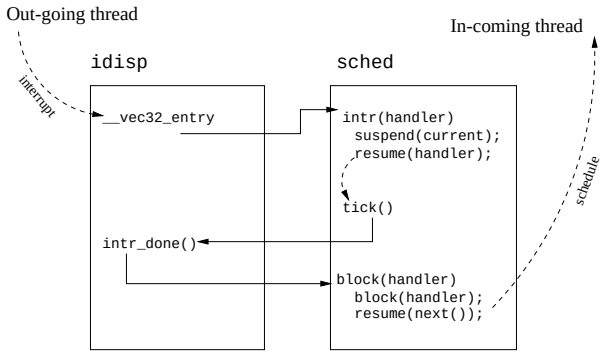

Figure 2: `sched` and `idisp` interaction

This mechanism results in extra temporal overhead due to the unnecessary resumption of the intermediate 'scheduler-thread' between thread switches. However, this mechanism means that the timer interrupt is handled just as any other interrupt. Furthermore, in a more complete system it is likely that `sched`'s `tick` method will perform useful work, such as checking for time-outs.

5 Experiments and Results

Two experiments were conducted to test the above implementation of decomposed preemptive scheduling. Firstly, a stress-test was conducted, to provide reasonable assurance of the scheme's (and implementation's) correctness. Secondly, performance tests have been conducted to ascertain the overhead imposed by the decomposition.

The experiments were conducted using an 90 MHz Intel Pentium P54C, with 32 MB of 90 ns EDO DRAM.

5.1 The Stress-Test

This test involved creating 1,000 threads, each thread (pseudo) randomly making an inter-component call or return, so that between 1 and 20 calls were nested in any one thread. In addition, key-strokes were produced randomly and processed by the command line in order to produce asynchronous stimuli for the system. This test ran continually for 48 hours and identified no bugs.

5.2 Performance Results

Each timer interrupt schedules a new thread, resulting in the interaction of 7 distinct components (the out-going and incoming thread and stack components, the interrupt-dispatcher, the scheduler and the ORB). Contrasting this with conventional systems (including micro-kernels) that incorporate these 7 distinct components into a single entity (namely, the kernel), it is clear that the above approach incurs potential performance problems.

An experiment was conducted to assess whether the decomposition of scheduling introduces unacceptable overhead. The experiment was conducted with two runnable threads, operating with interrupts disabled. A timer interrupt was simulated by issuing an `int 32` instruction from the first thread, and the time was measured for the second thread to receive control. Using the Pentium's `rdtsc` instruction to count processor cycles, this time was found to be just under 2,500 cycles.

With the default interrupt frequency of 100 ticks per second, this is a overhead of less than 0.3% on the 90MHz test machine used. Furthermore, relatively little of this overhead can be attributed to the decomposition. Firstly, the single interrupt causes 2 threads to be scheduled (the scheduler-thread, as well as the second runnable thread). Secondly, for simplicity the current implementation uses the Intel's TSS task-switching mechanism, known to perform poorly. Thirdly, the scheduling code is not heavily optimised.

Implementing these three optimisations is likely to drastically reduce the time.

6 Conclusions

This paper has described the implementation of preemptive scheduling on the Go! operating system. The scheduling mechanism described is decomposed into 5 types of component: the Go! ORB; an interrupt-dispatcher; the `stack` type; a `thread` type; and a scheduler.

Decomposing system services such as scheduling into constituent components offers several advantages, particularly to the next generation of embedded systems. Embedded systems must strike a balance between time-to-market, resource usage (including unit price and power consumption) and functionality. Decomposing system services allows the heart of embedded systems to be tailored more easily, reducing time-to-market and increasing functionality. Furthermore, the extremely low overheads of protection in Go! reduces resource consumption while also increasing flexibility (for example, the ability to 'hot-swap' a component's implementation without resetting the system).

Two of the main goals of this decomposition have been to improve systems-software engineering and configurability. In the authors' experience at least, significant improvements have been made in systems-software engineering. Improvements to configurability have been demonstrated by replacing component implementations without affecting other components. For example, a new implementation of the interrupt dispatcher that allows prioritisation of interrupts has been added, which does not affect any other components.

As well as the optimisations mentioned in Section 5, future work includes the development of a more complete system in order to demonstrate fully the benefits (or otherwise) of the techniques introduced here. Also, the experiences of other software-engineers will be valuable to asses Go!'s true impact to systems-software engineering.

References

[1] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An operating sytem architecture for application-level resource management. In *Proc. 15th Symposium on Operating System Principles (SOSP-95)*, pages 251–266, 1995.

[2] E. Gabber, J. Bruno, J. Brustoloni, A. Silberscatz, and C. Small. The Pebble Component-Based Operating System. In *Proc. 1999 USENIX Technical Conference, Monterey, CA*, June 1999.

[3] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), Seattle, WA*, pages 123–136, October 1996.

[4] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-Kernel-based systems. In *Proceedings of the 16th Symposium on Operating Systems*, pages 66–77, 1997.

[5] Trent Jaeger, Jochen Liedtke, Vsevolod Panteleenko, Yoonho Park, and Nayeem Islam. Security architecture for component-based operating systems. In *ACM Special Interest Group in Operating Systems (SIGOPS) European Workshop*, 1998.

[6] J. Keedy and J. Rosenberg. Support for objects in the MONADS architecture. In *Proc. Workshop on persistent object systems*, pages 202–213, Newcastle NSW (Australia), January 1989.

[7] Greg Law and Julie McCann. A New Protection Model for Component-Based Operating Systems. In *Proceedings of the IEEE Conference on Computing and Comminications, Phoenix, AZ, USA*, February 2000.

[8] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, February 1997.

[9] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, Series 2(42):230–265, 1936-1937.