PicOS: A Tiny Operating System for Extremely Small Embedded Platforms

E. Akhmetshina, P. Gburzynski and F. Vizeacoumar
Department of Computing Science
University of Alberta
Edmonton, Alberta CANADA T6G 2E8
{elvira, pawel, vizea}@cs.ualberta.ca

Abstract

We present a certain programming paradigm for implementing low-footprint applications on small embedded platforms and a tiny operating system based on that paradigm. The primary objective of our work was to create a friendly environment for rapid, reliable, and efficient deployment of customizable microcontroller applications primarily (but not necessarily) aimed at the wireless world. The proposed solution, while being characterized by very small resource requirements, offers an interesting flavor of multithreading and provides for well-structured self-documenting layout of the application code.

1 Introduction

Our work targets the smallest end of embedded applications, i.e., microcontrolled devices with small physical dimensions, small footprint (low-end CPU with little data memory), trivially low cost, and high durability (low battery consumption). It was directly inspired by a project whose objective was to develop a simple credit-card-size device equipped with a low-bandwidth short-range wireless transceiver. Using their transceivers, multiple devices present in the same area would be able to exchange some information. As most of the complexity of the device's behavior was related to the communication protocol, including the medium access (collision avoidance) scheme, a model of the target device was programmed in SMURPH [2, 4], which is a specification/modeling system for low-level communication protocols. The source code of the model, along with its English-language description, was then sent to the manufacturer for a physical implementation.

Some time later, the manufacturer sent us their program driving the device's microcontroller, for the purpose of assessing its conformance to our original specification. Striving to minimize the memory requirements of that program, the implementer organized it as a single thread executed on the bare CPU. Using an unintelligible combination of flags, hardware timers, and counters, the program tried to approximate the behavior of our high-level multi-threaded model. However, its original, clear, and self-documenting structure, consisting of a handful of simple threads presented as finite state machines [6], had completely disappeared in the process of implementation.

In our attempts to assist the manufacturer with the implementation of our model, we have concluded that indeed, using a "standard" (small) operating system [11, 10, 12] as a multithreaded execution platform would increase the application footprint beyond the capabilities of the originally assumed hardware design. However, we noticed that we could easily (and cheaply) implement a programming environment similar to the one offered by SMURPH, which would let us "port" or model almost directly to the actual hardware completely preserving its structure and verified properties.

This approach has brought us PicOS, which is a collection of functions for organizing multiple activities of "reactive" applications executed on a small CPU with limited resources. While some people may object to calling those tools an "operating system," they do provide a flavor of multitasking (implementable within very small RAM) as well as tools for inter-process communication. In our earlier work, we hinted at the possibility of using the SMURPH paradigm for building programs driving physical reactive systems [6]. In this paper, we demonstrate that the same paradigm can be applied to the design of operating systems for embedded platforms and highlight its advantages in this type of environment.

2 An overview

The primary problem with implementing classical multitasking on microcontrollers with limited RAM is minimizing the amount of fixed memory resources that must be allocated to a process/thread. One example of such a resource is the stack space, which must be preallocated (at least in some minimum safe amount) to every thread upon its creation. Even if the stack size per task is drastically limited, it still remains the most significant component of the total amount of memory resources needed to describe and sustain a single task. Additionally, the stack space is practically wasted from the viewpoint of the application:

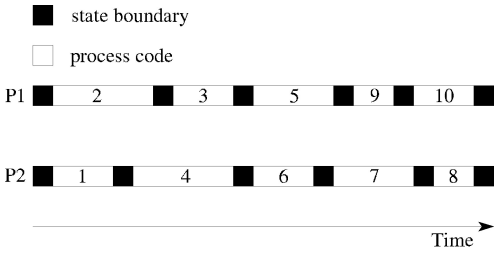

Figure 1: CPU multiplexing in PicOS: a possible execution order

it is merely a "working area" needed to build the high-level structure of the program, and it steals the scarce resource from where it is "truly" needed, i.e., for building the "proper" (global) data structures.

Our first implementation of PicOS, described in this paper, is for the Cyan Technology eCOG1 microcontroller [1] with 4KB on-chip RAM. Although it is possible to connect extra RAM to the microcontroller, our goal was to achieve non-trivial multithreading with no additional memory resources.

Consequently, the multitasking adopted by PicOS is not "classical." The different tasks share the same global stack and act as *coroutines* with multiple entry points and implicit control transfer. Each task looks like a finite state machine (FSM) that changes its states in response to events. The CPU is multiplexed among the multiple tasks, but only at state boundaries, as shown in Fig. 1. From the viewpoint of program execution, the FSM states can be viewed as *checkpoints* at which PicOS tasks can be preempted.

By a tradition (inherited from SMURPH), PicOS tasks are simply called *processes*. A process is described by its *code* (a C function organized in a special way) and data (representing the object on which the process is supposed to operate).

Processes are identifiable by integer numbers (process identifiers), assigned at the moment of their creation. A process is created explicitly (forked by another process) and can be *terminated* by itself or *killed* by another process. One process, called *root*, is started automatically by PicOS after reset. This process must be provided by the application and is responsible for creating all other processes. Some processes, typically device drivers, may be started internally by the system before the root process.

The special organization of the process code function (see Fig. 2) consists in introducing states into the code, which become multiple entry points via which the function can be entered. The states are identified by integer numbers (represented by symbolic constants in the example). Whenever a process is assigned the CPU, its code function is called in the current state, i.e., at one specific entry point. Initially, when the process is run for the first time after its creation, its code function is entered at state 0, which can be viewed as an initial state.

A running process may release the CPU (become blocked) indicating conditions (events) to resume it in the future. The latter is accomplished by issuing (explicit or implicit) *wait requests* before releasing the CPU. A wait request identifies an awaited event (or rather event type) and associates a state with that event. The simple idea is that the process will be resumed in the indicated state when the awaited event occurs. A blocked process may be waiting for more than one event at the same time. In such a case, it will be resumed by the first occurrence of *any* of those events. In particular, the order in which the corresponding wait requests were issued is immaterial. All the wait requests executed by the process in a given state aggregate to describe an unordered collection of awaited events.

Whenever a process is resumed and run, its collection of awaited events is cleared, i.e., the previously awaited (but possibly not triggered) events are forgotten. If the process decides to wait for the same or a similar configuration of events, it must reissue the respective wait requests from scratch.

If a process releases the CPU without issuing a single wait request, it will be unconditionally resumed in its last state, possibly after some other processes have been run in the meantime. This is equivalent to re-executing the current state from the beginning.

From the moment a process is resumed until it releases the CPU, it cannot lose the CPU to another process, although it may lose the CPU to the kernel, i.e., an interrupt. As shown in Fig. 1, the CPU is multiplexed among the multiple processes exclusively at state boundaries. The idea is that whenever the current process decides to go to sleep, the scheduler is free to allocate the CPU to another (available) process that is either not waiting for any event, or whose at least one awaited event is already pending.

3 Process example

Figure 2 shows fragments of a process code from a sample networking application programmed under PicOS. The language is straightforward C with operations like `process`, `endprocess`, `entry`, `release`, `proceed` implemented as macros.

The first argument of `process` provides the name of the process's code function, and the second argument specifies the type of the process's data object. Each process receives an implicitly declared local variable named `data` pointing to its data object. Thus, in Fig. 2, the type of `data` is `sess_t*`.

The argument of `entry` is a state number (typically, state numbers are represented by symbolic constants). The PicOS flavor of process preemption (and CPU multiplexing) is well illustrated by the sequence at state `RC_PASS`, where the process conditionally executes `wait` and `delay` followed by `release`. The first argument of `wait` is a numerical event identifier, also called a *signal*. Any number can be used for this purpose, with different

```
process (sniffer, sess_t)
  char c;
  entry (RC_TRY)
    data->packet = tcv_rnp (RC_TRY, efd);
    data->length = tcv_left (packet);
  entry (RC_PASS)
    if (data->user != US_READY) {
      wait (&data->user, RC_PASS);
      delay (1000, RC_LOCKED);
      release;
    }
    c = 1;
    ion (LEDS, CONTROL, &c, LEDS_CNTRL_SET);
 ...
  entry (RC_ENP)
    tcv_endp (data->packet);
    signal (&data->packet);
    proceed (RC_TRY);
endprocess (1)
```

Figure 2: A sample process

```
process (coordinator, bufstr_t)
  entry (MN_START)
    data->mon = fork (monitor, data->buf);
    for (int i = 0; i < data->N; i++) {
      fork (consumer, data->buf);
      fork (producer, data->buf);
    }
  entry (MN_WAIT)
    if (!running (consumer) ||
                   !running (producer)) {
      killall (producer);
      killall (consumer);
      kill (data->mon);
      terminate;
    }
    joinall (consumer, MN_WAIT);
    joinall (producer, MN_WAIT);
endprocess (1)
```

Figure 3: A sample coordinator for collaborating processes

numbers representing different events. It is a standard practice to use the address of the object (e.g., a lock or flag) to which the event is related. For `delay`, the first argument specifies the number of milliseconds after which a timer (alarm clock) event will be triggered. In both cases, the second argument identifies the state to be assumed by the process when the event occurs.

In our specific case, the process wants to be resumed at state `RC_PASS` when the awaited signal event is triggered, and at `RC_LOCKED` when the timer goes off, whichever happens first. The two operations are followed by `release`, which explicitly releases the CPU (and completes the execution of statements at the current state).

An event awaited by `wait` is triggered by `signal` (illustrated at state `RC_ENP`). The operation is ignored if the specified event is not awaited by any process; otherwise, it wakes up all processes waiting for the signal.

Local (automatic) variables declared by a process (like `c` in Fig. 2) are not preserved across state transitions. This is because all processes share the same stack. The process function is simply re-entered (called again) each time the process receives the CPU, in a specific state, which is selected as a `case` within a `switch` statement.

The implementation of blocking functions (e.g., system calls) is illustrated by the call to `tcv_rnp` in state `RC_TRY`. A function that is allowed to block accepts a state argument indicating where the process should be resumed when the function unblocks. Such a function may internally execute `wait` and `release`. PicOS follows a certain convention regarding the positioning of the state argument on the argument list of potentially blocking functions. If the function should be re-executed after the unblocking event (i.e., it unblocks on a new opportunity to "try again"), the state argument is the first item on the argument list. On the other hand, if the function unblocks after it has completed its job, the state argument appears last.

The role of `proceed` (used in state `RC_ENP`) is to perform an unconditional transition to the specified state. Unlike a simple "goto," this function crosses state boundaries and the process may lose the CPU while the transition is being made.

4 IPC tools

The system offers a carefully crafted collection of simple and orthogonal tools for implementing natural process collaboration scenarios. Some of those tools are illustrated by the sample process code listed in Fig. 3.

The process plays the role of a coordinator for a number of *producers*, a number of *consumers*, and a single *monitor* all accessing the same buffer (`data->buf`). A new process is started with the `fork` operation; its second argument identifies the data structure on which the created process will operate. Having started all processes, the coordinator transits to the second state, `MN_WAIT`, where it periodically checks if at least one producer and at least one consumer is still running. If not, the coordinator kills all the remaining processes of the set and also terminates itself. Otherwise, if at least one producer and one consumer is still present, the coordinator executes `joinall` to perceive the termination of any of the producers and/or consumers. Such an event will resume the coordinator in state `MN_WAIT`.

Note that standard synchronization tools, like semaphores or critical sections, are easily implementable using the `wait`/`signal` mechanism described in the previous section, in combination with simple flags or counters. Owing to the fact that PicOS processes are only preemptible at clearly identifiable checkpoints,

the intra-state chunks of code essentially execute as entryprocedures of a monitor. Consequently, potentially problematic operations on counters and flags, e.g., *Test and Set*, are always atomic, as long as they do not cross state boundaries.

Sometimes one would like to extend the FSM structure of a process onto a function invoked by the process, e.g., to let the function suspend itself waiting for an event, and then, when the event occurs, to continue from the point of suspension. This can be accomplished by turning the function into a process and "call" that process with the following sequence of operations:

```
join (fork (fun, data), state);
release;
```

available as this macro: `call (fun, data, state)`. The sequence forks a new process suspending the caller until the child terminates. The *state* argument identifies the state in the caller where it wants to be resumed when the child is done (returns).

5 Resource requirements

The minimum amount of memory needed to describe a single PicOS process is determined exclusively by the size of the Process Control Block (PCB) in the kernel. All processes (and also the kernel) share the same stack, which starts from the same level whenever a process function is entered.[1] Consequently, the number of processes in the system (the degree of multiprogramming) has no impact on the required minimum stack size, which is solely determined by the maximum configuration of nested function calls. As automatic variables are not very useful for processes (and thus discouraged), the stack has no tendency to run out. In the first version of PicOS, the stack size was set at 512 bytes, which value has never had to be increased and seems to be more than adequate for our most advanced applications.

The size of PCB is adjustable by a configuration parameter, depending on the number of events that a single process may want to await simultaneously. The standard setting of this limit is 3, which is sufficient for all our present applications (although 2 is not). The PCB size in bytes is equal to $8 + 4k$, where k is the maximum number of awaited events per process. For $k = 3$, this simple formula yields 20 bytes, which lets us describe more than 50 processes in 1KB of data RAM.

PicOS offers several configurable device drivers, including serial ports, LCD display, and LAN (Ethernet) interface. Those drivers are implemented as combinations of processes and asynchronous interrupt service routines. The Ethernet interface can operate in several modes including a special *wireless emulation mode* intended for emulating wireless channels through an *emulation server* running on a workstation connected to the local network. Even with all those devices configured in, the system can sensibly cater to non-trivial applications using no more than 4KB of data RAM available directly on the eCOG chip.[2]

The system is equipped with a memory allocator (`malloc/free`) capable of organizing the heap area into a number of disjoint pools. Under the conditions of tight memory, this solution is better than a single global memory pool because different components of the application are able to respond differently to allocation failures and, more importantly, the more critical components can be made independent of the greediness of the less critical ones. If additional data memory (SDRAM) is configured into the system, PicOS extends the area available to the heap and, additionally, provides functions for accessing the "spare" memory that cannot be directly mapped into the addressable data space.

6 Final remarks

The PicOS project started as a simple programming exercise inspired by our curiosity to see how far the modeling paradigm of SMURPH could be extrapolated onto real-life programs. The FSM structure and non-preemptibility of threads worked particularly well in SMURPH for the purpose of protocol modeling because 1) communication protocols are naturally amenable to this kind of description [8], 2) the non-preemptibility of intra-state code does not affect the accuracy of a discrete-time simulation model [2]. The FSM approach seems to be generally useful for programming reactive applications, i.e., ones whose primary role is to respond to events, rather than process data or crunch numbers [5], and can be viewed as a partial implementation of the StateCharts paradigm [7, 3]. It appears that most applications of small (tiny) embedded systems well fit this description and are thus easily expressible as sets of coroutines with multiple entry points.

One may frown, however, upon the non-preemptible character of PicOS threads, which compromises their responsiveness in real-time applications. Many people are inclined to believe that "embedded" implies "real-time," and what has not been designed as "real-time" from the bottom up cannot possibly be useful in the embedded world. Our observations suggest otherwise. Very few embedded applications pose non-trivial real-time requirements, and most of them are in fact able to put up with remarkable sloppiness and non-determinism in response time. One anecdotal study, in which one of us participated as a hired consultant, involved the design of a TV set-top box, with the project being announced as a real-time system development endeavor. Upon a closer scrutiny, it turned out that the sole "real-time" requirement of the target system was its capability to respond to clicks on the remote.

[1] Note that this also applies to processes "called" from other processes.

[2] This includes the 512B stack.

Notably, the non-preemptibility of threads in PicOS does not imply sloppiness in reacting to physical events. In particular, the interrupt service routines of PicOS are not constrained by thread scheduling, and they can handle interrupts at the full speed of hardware. The mechanism of synchronizing a thread (e.g., a device driver) to an interrupt event involves the standard `wait/signal` mechanism combined with a deterministic and trivially short lock. Our set of sample applications includes an Ethernet sniffer capable of intercepting all Ethernet frames in real-time, extracting information from their headers, and presenting that information in a formatted layout over the serial port. We are currently working on a plugin-extensible networking module implementing customizable wireless sessions, with TCP/IP being one of the general options. Nowhere in our work have we met with serious problems caused by the limited responsiveness or flexibility of PicOS processes.

To make a future version of PicOS better suited to real-time applications, we plan to introduce a hierarchy into the flat structure of processes by grouping them into *cohorts*. Cohorts will be preemptible, but all processes within a given cohort will still be treated as coroutines sharing the common stack. Processes belonging to different cohorts will synchronize (communicate) using special (restrictive) tools called *signal queues*. We believe that this approach will yield a reasonable compromise between the simplicity and flexibility understood as being able to accommodate hard timing constraints into PicOS.

One advantage of PicOS, which we learn to appreciate more and more, is the simplification (or rather elimination) of all thread synchronization problems, which seem to haunt application developers using traditional multithreaded platforms [9]. Note that in contrast to a naive and drastic solution, like making all methods in Java `synchronized`, the programmer-controlled granularity of states in PicOS still provides for a reasonable degree of concurrency and (adjustable) responsiveness. Additionally, regardless of their IPC patterns, PicOS processes are generally more reusable and self-documentable than traditional methods or functions. As state transitions are easily comprehensible and presentable in diagrams, the behavior of a well-designed PicOS process can usually be captured at a glance. Moreover, owing to the lack of synchronization problems in accessing shared data, PicOS processes are less dependent on data access patterns and, consequently, on their context. This makes it is easy to create libraries of processes, reuse processes, and adapt them to new functions.

C is obviously not the best programming language for PicOS processes. A few programming constructs built into the language would make PicOS programming easier and more natural than the present macros and ad-hoc tweaks. In particular, the state concept should be built into the language, instead of being emulated by `switch` and `case`. This would make it easier to introduce/identify states and also insert internal (hidden) states, e.g., used by functions called from a process. Additionally, a process code function should allocate its automatic variables statically or within the process data area (e.g., depending on a keyword) but not on the stack, which is often confusing.

In summary, our observations hint at the idiosyncratic character of small embedded systems regarding the right shape and size of their software development platforms. What is considered natural and friendly from the viewpoint of a large (classical) computer system looks like an overkill when applied to a small system expected to cater to a restricted class of custom applications. The programming paradigm of PicOS seems to well match the requirements of simplicity, reliability, and flexibility posed by contemporary microcontroller applications.

References

[1] Cyan Technology. *eCOG1/eCOG1i 16-bit Processor, version 2.0*, 2002. See also: `http://www.cyantechnology.com/`.

[2] W. Dobosiewicz and P. Gburzynski. Protocol design in SMURPH. In J. Walrand and K. Bagchi, editors, *State of the art in Performance Modeling and Simulation*, pages 255–274. Gordon and Breach, 1997.

[3] J. Gamble. *Towards a Methodology for Intelligent Activity Monitoring*. University of Alberta, Department of Computing Science, Edmonton, Canada, 2001. MSc Thesis.

[4] P. Gburzyński. *Protocol design for local and metropolitan area networks*. Prentice-Hall, 1996.

[5] P. Gburzyński and J. Maitan. Simulation and control of reactive systems. In *Proceedings of Winter Simulation Conference WSC'97*, pages 413–420, Atlanta, Georgia, Dec. 1997.

[6] P. Gburzyński and J. Maitan. Specifying control programs for reactive systems. In *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA'98*, pages 1702–1709, Las Vegas, July 1998.

[7] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts*. McGraw-Hill, 1998.

[8] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall Int., 1991.

[9] B. Kauler. *Flow Design for Embedded Systems*. R&D Books, 1997.

[10] QNX Software Systems. *QNX Neutrino OS System Architecture Guide, version 6.2*, 2002.

[11] Wind River Systems. *VxWorks 5.x Datasheet*, 2002.

[12] K. Zuberi and K. Shin. EMERALDS: a small-memory real-time microkernel. *IEEE Transactions on Software Engineering*, 27(10):909–928, Oct. 2001.