
Proceedings Paper:
Trenkwalder, S.M., Lopes, Y.K., Kolling, A. et al. (3 more authors) (2016) OpenSwarm: an event-driven embedded operating system for miniature robots. In: Proceedings of IROS 2016. 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems, 09-14 Oct 2016, Daejeon, Korea. IEEE . ISBN 978-1-5090-3762-9

https://doi.org/10.1109/IROS.2016.7759660

OpenSwarm: An Event-Driven Embedded Operating System for Miniature Robots

Stefan M. Trenkwalder[1], Yuri Kaszubowski Lopes[1], Andreas Kolling[1],
Anders Lyhne Christensen[2], Radu Prodan[3] and Roderich Groß[1]

***Abstract*— This paper presents OpenSwarm, a lightweight easy-to-use open-source operating system. To our knowledge, it is the first operating system designed for and deployed on miniature robots. OpenSwarm operates directly on a robot's microcontroller. It has a memory footprint of 1 kB RAM and 12 kB ROM. OpenSwarm enables a robot to execute multiple processes simultaneously. It provides a hybrid kernel that natively supports preemptive and cooperative scheduling, making it suitable for both computationally intensive and swiftly responsive robotics tasks. OpenSwarm provides hardware abstractions to rapidly develop and test platform-independent code.**

We show how OpenSwarm can be used to solve a canonical problem in swarm robotics—clustering a collection of dispersed objects. We report experiments, conducted with five e-puck mobile robots, that show that an OpenSwarm implementation performs as good as a hardware-near implementation. The primary goal of OpenSwarm is to make robots with severely constrained hardware more accessible, which may help such systems to be deployed in real-world applications.

I. INTRODUCTION

Swarm robotics investigates the emergence of collective behavior through local interactions of simple robots in large groups (swarms) [1]. Due to the large number of robots within a swarm, each robot typically has to be inexpensive and is designed to rely on basic operations. As a result, many swarm robots have severely limited on-board resources and can be categorized as Class 0 or Class 1 constrained devices [2]. For instance, the Kilobot [3] platform is a Class 0 constrained device by providing an Atmel ATmega328P with 2 kB RAM and 32 kB ROM.

Due to the limited on-board resources, swarm robotics algorithms are, in general, implemented at a low level (hardware-near). Consequently, their implementation cannot be directly ported to other platforms. They are usually onerous to maintain or extend, and are likely to contain programming errors [4]. This resembles the beginnings of personal computers, where every machine executed only a single program—such as a word processor or accounting software. With the advent of operating systems and concurrency models, complex software could be developed more effectively and personal computers could execute multiple applications concurrently. As a result, personal computers became easier to use and program, and, hence, they were used more widely.

The benefits of operating systems have already led to the development of embedded operating systems in the field of sensor networks [5] and the Internet of Things [6]. Sensor network motes[1] and Internet of Things devices provide limited resources similar to swarm robots. For instance, the sensor mote AS-XM1000, a Class 1 constrained device, provides 8 kB RAM and 116 kB ROM. Embedded operating systems—such as Contiki [7] and TinyOS [8]—manage these resources and facilitate application development. They are mainly designed to send sensor data to a central base unit, which requires them to execute short sequences of instructions. However, robots may also need to execute computationally expensive algorithms—for example, image processing. As a result, operating systems for robots have different requirements.

This paper introduces OpenSwarm, an open-source, event-driven operating system for miniature robots. It provides a novel hybrid kernel that, unlike other kernels for microcontroller-based systems, natively supports two concurrency models, preemptive and cooperative scheduling. It uses events as hardware-abstraction to interface components of the system with hardware. OpenSwarm provides mechanisms to restrict, convert, and manipulate sensor and actuator values for further—higher levels of—abstraction. All this makes algorithms and code easier to port, maintain, and extend.

We compare OpenSwarm to state-of-the-art embedded operating systems. OpenSwarm's static and dynamic memory footprint is analysed. OpenSwarm is then used to solve a canonical problem in swarm robotics on the e-puck miniature mobile robot [9], a Class 1 constrained device. The performance of the OpenSwarm implementation is compared with that of a hardware-near implementation presented in [10].

The paper is structured as follows. Section II summarizes existing systems that influenced OpenSwarm. The OpenSwarm architecture is presented in Section III. Section IV details its implementation and its evaluation is presented in Section V. Section VI concludes the paper.

II. RELATED WORK

The need for easy-to-use software environments in robotics resulted in a variety of middlewares [11]. They are

[1]S.M. Trenkwalder, Y.K. Lopes, A. Kolling, and R. Groß are with the Department of Automatic Control and Systems Engineering, The University of Sheffield, Sheffield, UK, e-mail: {s.trenkwalder, y.kaszubowski, a.kolling, r.gross}@sheffield.ac.uk

[2]A.L. Christensen is with the BioMachines Lab, University Institute of Lisbon, (ISCTE-IUL), Lisbon, Portugal, e-mail: alcen@iscte.pt

[3]R. Prodan is with the Institute of Computer Science, University of Innsbruck, Innsbruck, Austria, e-mail: radu@dps.uibk.ac.at

[1]A mote is a sensor device inside a sensor network.

designed to simplify development, but require an underlying operating system. One of the most common middlewares is ROS [12]. Small units of ROS, which are called *ROS nodes*, can be executed on small and embedded devices [13], but need a ROS master and core functions, which are commonly executed on a central device capable of running Linux. Consequently, ROS and other robot middlewares are not suitable for scalable swarm robotics applications.

Some frameworks—such as ASEBA [14] and JaMOS [15] can be seen as interpreters implemented on miniature robots. ASEBA is a virtual machine that executes a hardware-independent script language. This improves the portability and maintainability. However, the performance is limited (on average, one instruction of virtualized code translates to 70 processor instructions [14]). JaMOS is a finite state machine operating system, which uses an XML-based motion description language (MDL2ε). This provides good reusability of code. Both systems—ASEBA and JaMOS—only provide predefined sets of functions, which restrict the verity of possible solutions.

Wireless sensor networks, which work under similar computational constraints as swarm robotics systems, already make use of concepts of operating systems [5]. Operating systems for sensor network motes manage resources, including memory, sensors, and timers. With growing complexity of applications, these operating systems provide different ways to execute one or more programs in a controlled manner.

In general, there are two commonly used execution models in operating systems: preemptive and cooperative. Preemptive scheduling interrupts the executing thread (or process) after a certain execution time, stores its current state, and changes to another thread. It executes multiple threads—each for a short time. It requires one call stack per thread and consumes more time and memory than cooperative scheduling. However, preemptive scheduling is better suited to execute computationally intensive algorithms.

Cooperative scheduling typically uses functions or processes that run to completion. Commonly, they are triggered by events. Short sequences of code are more efficiently scheduled cooperatively than preemptively. However, long sequences of code tend to monopolize computational resources and can cause execution delays. Furthermore, cooperative scheduled code, in general, is difficult to maintain and debug.

TinyOS [8] and Contiki [7] are the most common embedded operating systems in sensor networks and are both designed to provide a cooperative execution model and events. Additional functions to provide software threads are also provided. However, their thread-models have drawbacks. TinyOS provides TOSThreads, which are executed in a single high-priority kernel thread. They are only executed when TinyOS becomes idle [16]. As a result, they have a lower priority than the cooperative execution model. Contiki provides Protothreads [17], which are two-byte-sized stackless threads that cannot be preempted in a common time-slicing context. Each Protothread yields cooperatively and another thread can take over [18]. A thread is only preempted by a hardware interrupt which executes certain run-to-completion handler functions. As a result, Contiki is not a preemptive operating system according to the definition in [19].

The execution models of LiteOS [20], Mantis OS [21], Nano-RK [22] and NuttX [23]—four embedded operating systems—are based on preempted threads, which require one call stack per thread. This creates a memory and time overhead compared to cooperative scheduling. LiteOS and Mantis OS use a priority-based round-robin scheduler. NuttX uses FIFO/priority-based round-robin and sporadic scheduling. Nano-RK uses rate-harmonized and rate-monotonic scheduling, which provides real-time properties.

The aforementioned operating systems are mainly designed to support measurements and communication, which results in different system requirements compared to miniature robots. For instance, motes typically have long inactive and short active times. They are designed for low energy consumption. On the other hand, a robot might constantly interact with its environment. Consequently, it cannot be assumed that the robot has long inactive periods. Interactions—such as locomotion and actuation—usually require a significantly higher amount of energy than the processor unit. Implicit assumptions during the design of the presented operating systems—such as processing time, energy consumption and the lack of actuation—make these systems difficult to adapt to the needs in robotics.

OpenSwarm, presented in this paper, has been primarily designed for swarm robotics. It provides the benefits of operating systems as well as an easy-to-use software environment. It combines a preemptive execution model of systems like LiteOS or Mantis OS with a cooperative execution model, similar to TinyOS. The resulting hybrid kernel enables OpenSwarm to compute long computational-intensive tasks (such as image processing) at the same time as short swiftly-reacting tasks (such as obstacle avoidance). OpenSwarm also provides an architecture to manage input and output devices (such as actuators) in a device-independent manner.

III. ARCHITECTURE

OpenSwarm provides a monolithic kernel[2], which allows a small memory footprint and a high performance [19]. It is designed for platforms that lack support for different access privileges (such as user and kernel mode) or memory protection—for example, via a memory management unit (MMU).

To provide an efficient system, user applications and OpenSwarm are compiled into a single image, which is uploaded to the microcontroller. One benefit of this approach is that an application uses system calls directly and thus generic translation tables are not needed.

Figure 1 shows the architecture of OpenSwarm. User processes (applications) are hardware-independently executed

[2]A monolithic kernel compiles to one executable, is not layered, and contains no un-/loadable modules [19].

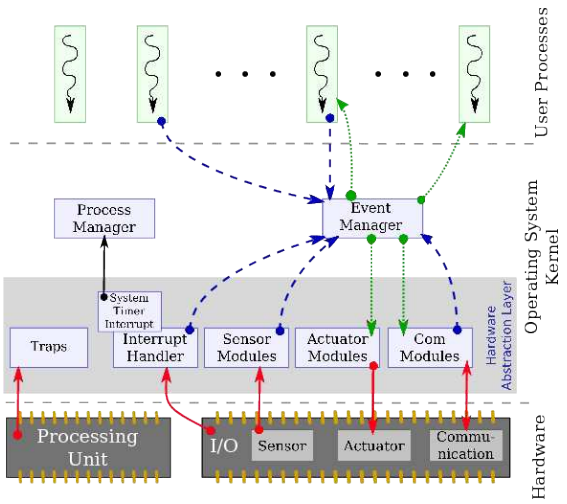


Fig. 1: The OpenSwarm architecture. The top layer comprises the user processes (applications). The middle layer represents the monolithic kernel. The bottom layer comprises the hardware. Green boxes represent single-threaded processes running in a flat memory space. Blue boxes indicate modules of the kernel. Solid lines represent direct interactions with modules, where red lines indicate hardware dependent interactions. Dashed blue lines indicate interactions through emitting events. Dotted green lines represent the execution of events through handlers or previously-executed blocking functions.

on top of the kernel. The kernel is thus the interface between applications and hardware. It is divided into three logical units: the Process Manager to provide concurrency, the Event Manager to exchange information, and the Hardware Abstraction Layer (HAL) for hardware abstraction.

A. Process Manager Module

OpenSwarm provides a novel hybrid kernel that natively supports both preemptive and cooperative scheduling. OpenSwarm is hence a multi-tasking system, that is, it can execute multiple processes within a certain period concurrently. A task can either be a process or a cooperatively scheduled event handler.

A process is an independently-managed sequence of instructions with its own call stack. Processes are executed for a guaranteed time interval (10 ms), which is controlled by a time-slicing preemptive scheduler. In OpenSwarm, processes are single-threaded. Due to the lack of separate protected memory space, they behave like threads to each other in a flat memory space. Therefore, *thread* and *process* are used interchangeably in this paper.

OpenSwarm uses a five-state model where a process can be `new`, `ready`, `running`, `blocked`, or `zombie` (see Figure 2). During creation, a process is `new` and, during termination, it is `zombie`. During its life-cycle, a process is in one of the states `ready`, `running`, or `blocked`. The change from one running process to the next is done by the scheduler.

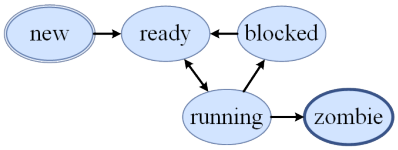


Fig. 2: OpenSwarm's process state model. While a process is being created, it is in state `new` (initial state). As soon as the process is ready to be executed, it changes to `ready`. The currently executed process is in state `running`. `blocked` indicates that a process is waiting for an event to occur. Once the event has occurred, the process changes its state to `ready`. State `zombie` (final state) indicates that a process is being terminated.

The scheduler stores and loads all working registers of the processing unit and the call stack. Its algorithm can be changed at run time. This can be used to implement schedulers with different properties—such as real-time. By default, a round-robin scheduler is used to guarantee fairness.

For short reactive sequences of code, cooperative scheduling is preferable. It can be achieved by subscribing handler functions to an event. Once this event occurs, the handler function is executed and runs to completion. The handler cannot use blocking functions, because it is not executed as part of a thread and does not have its own call stack. However, it still can be preempted by hardware interrupts.

B. Event Manager Module

In OpenSwarm, information between processes or modules is exchanged through events. Events provide platform-independent abstraction to increase the portability of user code. They can be used to access hardware functions (e.g. obtain sensor values). To use an event, first, an event (such as "*new camera frame*") has to be registered, which informs the operating system that this event can occur. When an event is send and subsequently received, it is stored and processed between the execution of threads to not compromise the execution time of the currently running thread. If multiple events occurred, each event is processed sequentially.

Events can also be used for interprocess communication (IPC) and synchronization. Synchronization can be achieved by calling blocking functions, which unblock once an event has occurred and the transferred information matches a user-defined condition.

C. Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) interfaces the robot's hardware with the hardware-independent part of the system, that is, the processes and events. The HAL is modular in design. Each input/output (I/O) device is represented by a dedicated module, which contains two layers (see Figure 3). The lower layer interacts directly with the hardware via a device-specific handler (including interrupts and registers). If needed, this layer can also use other modules—such as I^2C or SPI. The top layer transforms values from hardware-specific to hardware-independent, and vice versa, via a *processor*.

Processors are callback functions that can be changed during run time by registering different functions. Note that they can also be used to implement virtual sensors, virtual actuators, or safety measures. For the case study in this paper, we use a virtual line-of-sight sensor, which is realized using a camera (see Section V-B). Virtual actuators could be used on an omnidirectional robot to emulate different kinematic models. Safety measures could impose behavioural restrictions similar to a subsumption architecture [24].

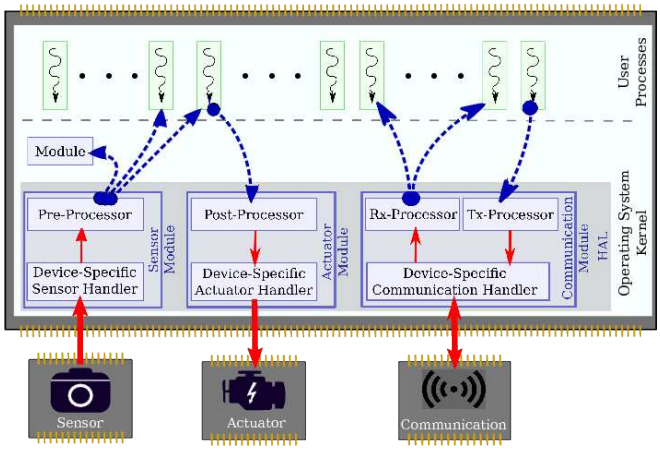


Fig. 3: Processing of data to/from input, output, and communication devices in OpenSwarm. Dashed blue lines represent events. Red arrows represent hardware-specific data transfer. Data exchange between different devices is indicated by thick red arrows. Similar to Fig. 1, blue and green boxes represent modules and processes of OpenSwarm.

HAL also comprises modules for hardware traps[3] and interrupts. One notable interrupt enables periodic scheduling of threads. It is called the *system timer interrupt* (see Figure 1).

In general, HAL interfaces three types of devices: input, output, and communication devices.

1) Input: Input devices, such as sensors, communicate with the microcontroller via binary input or analog-to-digital converter (ADC). After obtaining a value from the hardware by executing the device-specific handler, the Pre-Processor transforms it into a hardware-independent value and sends the result as an event to all subscribers, as illustrated on the left side of Figure 3. For instance, a proximity sensor reading of 3520 obtained by an ADC might be transformed to 5 mm. Non-linear sensor values could be linearized and converted to arbitrary units.

2) Output: Output devices, such as actuators, interact with the microcontroller via binary output or digital-to-analog converter (DAC). Once an actuation value has been sent to the Actuator Module, this value is then transformed by the Post-Processor to a hardware-specific value and applied to the hardware by the device-specific handler (see Figure 3). For example, consider a robot that shall move with a certain velocity in a particular direction. First, a thread emits an event that contains the desired hardware-independent direction and velocity (e.g. forward with 0.2 $\frac{\text{m}}{\text{s}}$). It is then converted into hardware-dependent values by the Post-Processor. Under normal circumstances, the robot would immediately apply these values to its actuators. However, let us assume that the Actuator Module received an event informing it of an unsafe area in front of the robot. The Post-Processor could then change the hardware-dependent values to prevent the robot from entering the unsafe area.

3) Communication: Communication devices, such as I^2C or UART, are commonly embedded in microcontrollers. The structure to implement a communication device is similar to the I/O modules. Raw communication messages, once obtained by the device-specific handler, are unpacked by the Rx-Processor. Error detection or correction is also applied according to an implemented protocol. The retrieved data is subsequently sent as an event to all subscribers. Once an event has been received by the Tx-Processor, outgoing data is dynamically buffered and packed into a message. Thereafter, it is sent asynchronously by the hardware-specific handler. Note that different protocols can be implemented by extending or replacing the Rx-/Tx-Processors.

IV. IMPLEMENTATION

OpenSwarm is released under an open-source license (adapted FreeBSD License) and its source code can be found at [25]. Documentation, tutorials, and further details can be found at www.openswarm.org. OpenSwarm is written in C to facilitate the portability to other devices. The used implementation supports the miniature mobile robot e-puck[4] [9].

Hardware-specific code was kept to a minimum (489 of 2783 lines of code = 17.6 % of OpenSwarm v0.15.9.15 without I/O device implementations). To deploy OpenSwarm onto another robot platform, only a subset of OpenSwarm's functions have to be reimplemented. An example of how to design and port a module of OpenSwarm to the Kilobot platform [3] is provided in [27].

OpenSwarm supports the inclusion of new HAL modules, enabling the extension of its features. It is good practice to avoid access to low-level hardware from the user space as this would reduce the portability of the code. However, developers could access, for instance, raw sensor data or other low-level hardware resources.

A. System Calls

OpenSwarm provides system calls to execute, manage, communicate with, and synchronize threads. Table I lists the most important system calls required to write applications.

1) Initialize and Start: OpenSwarm is made operational by calling `Sys_Init_Kernel` and `Sys_Start_Kernel`. `Sys_Init_Kernel` initializes the input and output ports, the system timer, controllers of periphery, and communication. This also generates the system thread. `Sys_Start`

[3]Hardware traps are detected fault states of the processing unit.

[4]To facilitate the readability of the source code, three source files (`e_epuck_ports.h`, `e_init_port.h`, `e_init_port.c`) from the e-puck library [26] were used to define I/O registers and configure the robot.

TABLE I: A list of important system calls of OpenSwarm. System calls can be used in threads. A complete list can be found in [28].

System Call	Description
Sys_Init_Kernel Sys_Start_Kernel Sys_Run_SystemThread	Initialize and Start
Sys_Start_Process Sys_Kill_Process Sys_Yield Sys_Start_CriticalSection Sys_End_CriticalSection	Task Management
Sys_Register_Event Sys_Unregister_Event Sys_IsEventRegistered Sys_Subscribe_to_Event Sys_Unsubscribe_from_Event	Event Management
Sys_Send_Event Sys_Wait_for_Event Sys_Wait_for_Condition	Interprocess Communication & Synchronization

_Kernel starts all necessary timers and enables all interrupts. The latter two functions were separated to give the user the option to execute code before system start. This would allow, for instance, events to be registered before any thread is executed.

Once OpenSwarm has been started, the system thread is executed by calling Sys_Run_SystemThread.

2) Task Management: To add new threads, Sys_Start_Process allocates the process control block (PCB) [29] and thread-specific stack. Once the thread is due, the processing unit executes the function that was passed as a parameter. Each thread has a unique identifier, *processID*, and can be terminated and removed with Sys_Kill_Process. If a process wants to suspend itself prematurely, it calls Sys_Yield.

Running threads may contain a critical section of code that needs protection from interruption to prevent data inconsistencies or program faults. Sys_Start_CriticalSection and Sys_End_CriticalSection start and end a critical section, respectively. If the system timer interrupt occurs, scheduling is postponed to the end of the critical section. It is worth noting that critical sections could also prevent the change of threads and, therefore, can monopolize the processing unit, if used excessively.

3) Event Management: To use an event, it must first be registered in the system by using Sys_Register_Event. Similarly, Sys_Unregister_Event is used to remove an event. Sys_IsEventRegistered checks if an event is registered. A complete list of available events can be found in [27], [28]. Furthermore, threads subscribe a certain handler function (callback function) to an event by executing Sys_Subscribe_to_Event. Sys_Unsubscribe_from_Event unsubscribes a thread. Note that event handler functions can only be subscribed once to a specific event by a specific thread. However, different threads can subscribe the same handler to the same or different events.

4) Interprocess Communication & Synchronization: To communicate with other threads, data can be send with the non-blocking function Sys_Send_Event. The transferred data is copied to a process-internal event table and processed asynchronously by the Event Manager. All event handler functions are executed between two scheduled threads. This provides a fast response time without delaying the running thread.

Concurrent threads can be synchronized by Sys_Wait_for_Event, which blocks a thread until a specific event occurs and returns a pointer to its data structure. Sys_Wait_for_Condition blocks a thread until the event data meets a condition defined by a function that was passed as a parameter. The condition function can represent user specific requirements and does not have to process the transferred data. For instance, it can wait until a 1 ms clock event occurred 250 times, if the application has to wait 250 ms.

V. EVALUATION

The evaluation is based on OpenSwarm v0.15.9.15 compiled by Microchip MPLAB XC16 Compiler v1.25 (released 28.07.2015 for Linux 64-bit) for the miniature mobile robot e-puck [9]. The e-puck was chosen due to its wide use in the domain of swarm robotics [30], [31], [32]. Its processing unit is the Microchip dsPIC30F6014A, which provides 15 MIPS, 8 kB RAM, and 144 kB ROM.

A. General Properties

Table II shows general properties of OpenSwarm and other embedded operating systems. As can be seen, OpenSwarm is the only embedded operating system to natively support both cooperative and preemptive scheduling. Scheduling mechanisms that are natively implemented in the system are more efficient than mechanisms marked as 'library', as the latter are executed in the application layer. All operating systems provide resource sharing mechanisms. The two operating systems that support threads through additional software library and cooperative scheduling (i.e. Contiki and TinyOS) do not provide system-wide process synchronization mechanisms.

OpenSwarm has an overall static memory footprint of 1 kB of RAM and 12 kB ROM. Based on the memory footprints listed in Table II, OpenSwarm consumes a relatively small amount of memory. The memory footprint values for other systems are from the literature [33], [7], [21], [22], [20], [23]. These values result from implementations on different devices and may change for other devices.

IN OpenSwarm, each thread requires 146 B[5] of dynamically allocated memory. Each event uses 6 B of dynamically allocated memory. In addition, the data that was sent last with event i is also buffered and consumes memory (d_i B). Overall, the required RAM (in bytes) for OpenSwarm v0.15.9.15 can be calculated for n user threads and m user events by:

$$1142 + 146n + 6m + \sum_{i=1}^{m} d_i. \quad (1)$$

[5]This assumes a default stack size of 128 B. The required memory increases if the user allocates memory inside the thread.

TABLE II: List of embedded operating systems and key properties.

Operating System	Architecture	Programming Model	Scheduling		Events	Dynamic Memory	Resource Sharing	Process Synch.	Memory Usage	
			Cooperative	Preemptive					RAM	ROM
Contiki [7]	modular	events & interrupts	natively	only with interrupts	yes	yes	serial access	no	2 kB	60 kB
LiteOS [20]	modular	threads	no	natively	yes	yes	mutex	yes	$<$ 4 kB	$<$ 128 kB
Mantis OS [21]	layered	threads	no	natively	no	yes	semaphores	yes	500 B	14 kB
Nano-RK [22]	monolithic	threads	no	natively	only to synchronize	no	mutex & semaphores	yes	2 kB	18 kB
NuttX [23]	modular	threads	no	natively	no	yes	POSIX standard[a]	yes	n/a	$\geq$ 32 kB[b]
TinyOS [8]	monolithic	events & threads	natively	library	yes	no	virtualization & events	no[c]	n/a	$\geq$ 400 B[d]
OpenSwarm	**monolithic**	**events & threads**	**natively**	**natively**	**yes**	**yes**	**event-based IPC**	**yes**	**1 kB**	**12 kB**

[a]NuttX provides POSIX standard Interprocess Communication methods, such as semaphores, shared memory, or message queues [23].
[b]NuttX requires 32 kB in its smallest configuration [23]. The memory footprint of a deployed system is not documented.
[c]TinyOS cannot synchronize threads with the event-driven cooperative part of it. Between threads, processes can synchronize.
[d]TinyOS's core elements use 400 B [8]. The memory footprint of a deployed system is not documented.

B. Swarm Robotics Case Study

We use a canonical swarm robotics task, object clustering [10], to illustrate the use and analyse the performance of OpenSwarm. The task requires a group of robots to push together objects. Each robot uses a line-of-sight sensor to determine what is in front of it: another robot, an object, or nothing (i.e. a wall, if the environment is bounded). The detection is based on color: robots are green, objects are red, and the environment is white. The controller maps the obtained color onto a pair of predefined wheel speeds [10].

1) Implementation: Listing 1 shows the controller implementation using OpenSwarm. To realise the line-of-sight sensor, the image from the on-board camera is transformed into a single color value by a customized pre-processor (not shown). The resulting value is then sent as a `SYS_IO_1PXSENSOR` event. Note that the transformation to a virtual line-of-sight sensor requires a significant amount of the available resources.

Function `main` initializes the system. It registers all system events—including `SYS_[LEFT|RIGHT]MOTOR_SPEED`, `SYS_SEND_UART1`, `SYS_1ms_TIMER`, and `SYS_IO_1PXSENSOR`. Then, the object clustering controller (`object_clustering`) is subscribed to the camera event `SYS_IO_1PXSENSOR`. A thread `loggingThread` is allocated. OpenSwarm is then started, including timers and interrupts. From that point onwards, the operating system (system thread) and the logging thread are executed concurrently.

In parallel to threads, the `object_clustering` controller is cooperatively executed each time event `SYS_IO_1PXSENSOR` occurs. Depending on the received color value, the appropriate speed values are send to the motors with events (`SYS_[LEFT|RIGHT]MOTOR_SPEED`). The wheel speeds parameters `ROBOT_SPEED_[L|R]`, `OBJECT_SPEED_[L|R]`, and `NOTHING_SPEED_[L|R]` are taken from [10].

The `loggingThread` sends the system time and the wheel speeds via Bluetooth (connected to UART1). After the execution, the thread waits 250 ms by using the `SYS_1ms_TIMER` event with the `wait250times` condition function.

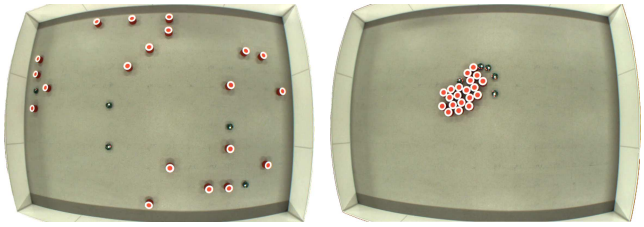
```
1  void loggingThread();
2  void wait250time(sys_event_data *data);
3  bool object_clustering(uint16 PID, uint16 ↩
       EventID, sys_event_data *data);
4
5  int main(void){
6    Sys_Init_Kernel();//initialise OS
7
8    Sys_Subscribe_to_Event(SYS_IO_1PXSENSOR, 0,↩
         object_clustering, NULL);
9    System_Start_Process(loggingThread);
10
11   Sys_Start_Kernel();//start OS
12   Sys_Run_SystemThread(); //run system thread
13 }
14
15 bool object_clustering(uint16 PID, uint16 ↩
       EventID, sys_event_data *data){
16   sys_colour rx_color = data->value;
17
18   switch(rx_color){
19   case GREEN://Other robot
20     Sys_Send_IntEvent(SYS_LEFTMOTOR_SPEED, ↩
           ROBOT_SPEED_L);
21     Sys_Send_IntEvent(SYS_RIGHTMOTOR_SPEED,↩
           ROBOT_SPEED_R);
22     break;
23   case RED://object
24     Sys_Send_IntEvent(SYS_LEFTMOTOR_SPEED, ↩
           OBJECT_SPEED_L);
25     Sys_Send_IntEvent(SYS_RIGHTMOTOR_SPEED, ↩
           OBJECT_SPEED_R);
26     break;
27   case WHITE://nothing or wall
28     Sys_Send_IntEvent(SYS_LEFTMOTOR_SPEED, ↩
           NOTHING_SPEED_L);
29     Sys_Send_IntEvent(SYS_RIGHTMOTOR_SPEED, ↩
           NOTHING_SPEED_R);
30     break;
31   default://anything else
32     break;//do nothing
33 } }
34
35 void loggingThread(){
36   static char message[24];
37   while(true){
38     sprintf(message, "\%u:(\%u,\%u)\n", ↩
           Sys_Get_SystemTime(), ↩
```

```
            Sys_Get_MotorSpeed_Left(), ↩
            Sys_Get_MotorSpeed_Right());
39      Sys_Send_Event(SYS_SEND_UART1, message, ↩
            24);//send via Bluetooth
40
41      Sys_event_data  data = ↩
            Sys_Wait_for_Condition(SYS_1ms_TIMER,↩
             wait250times);
42      Sys_Clear_EventData(&data);
43  } }
44
45  void wait250time(sys_event_data *data){
46    static uint8 counter = 0;
47    if(counter++ < 250){
48      return false;
49    }
50    counter = 0;
51    return true;
52  }
```

Listing 1: Program used in the object clustering case study. This code illustrates how OpenSwarm can be used. It was executed on each of the five physical e-pucks in the case study.

2) Experiment Setup: In the experiment, five robots pushed 20 objects together to form a cluster (see Figure 4). The robots and objects were uniformly randomly distributed on 165 pencil marks arranged in a 15×11 grid. The pencil marks are equally spaced (25 cm) in a 400×300 cm arena with light grey floor surrounded by 50 cm tall white walls. The robots were fitted with green skirts to simplify identification. The objects were red and had a diameter of 10 cm and a height of 10 cm.

Twenty experimental trials were conducted, ten with the hardware-near [10] and ten with the OpenSwarm implementation. The experiments were conducted following the same protocol as in [10]. Each trial lasted 15 minutes. Video recordings of all experiments are available on the online supplementary material website [27].



(a) (b)

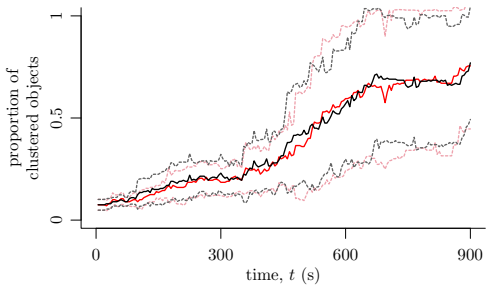
Fig. 4: Snapshots of the object clustering experiment: (a) initial distribution of robots and objects; (b) distribution after 10 min.

To provide comparable results, OpenSwarm and the hardware-near implementation used the same camera configuration and logic to obtain the sensor values and the logging thread was disabled. Due to sensor misalignment and noise, the line-of-sight sensor value was calculated from a 160×160 sub-frame of the 640×480 CMOS RGB camera with a $8\times$ digital zoom. This created reliable sensor values within 150 cm [10].

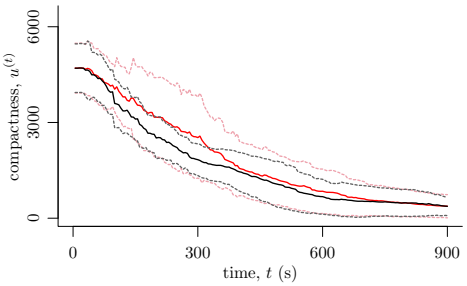
3) Analysis: Figure 5 shows results of the hardware-near implementation (black) and the implementation with OpenSwarm (red). We consider two performance metrics [10]. Figure 5a shows the proportion of objects in the largest cluster (a) and the compactness of the cluster (b). The compactness is computed, as in [10], by

$$u^{(t)} = \frac{1}{4r_o^2} \sum_{i=1}^{N_o} ||\mathbf{p}_i^{(t)} - \bar{\mathbf{p}}^{(t)}||^2, \tag{2}$$

where r_o is the radius of the object, N_o is the number of objects, $\mathbf{p}_i^{(t)}$ denotes the position of object i, and $\bar{\mathbf{p}}^{(t)}$ represents the centroid of the center of all objects.



(a)



(b)

Fig. 5: Dynamics of object clustering with five physical robots and 20 objects, (a) average proportion of clustered objects over time; (b) average compactness of objects over time. In black, the results obtained in 15 minutes using the hardware-near implementation [10]. In red, the results obtained in 15 minutes using OpenSwarm. The dashed grey and light-red lines represent the mean value $\pm$ standard deviation of the hardware-near and OpenSwarm implementation, respectively.

Overall, both the hardware-near and OpenSwarm implementations are similar in performance (see Figure 5). Both implementations succeeded in clustering the objects.

VI. CONCLUSION

We presented OpenSwarm, the first operating system designed for severely computationally constrained robotic systems. OpenSwarm provides a multi-tasking environment, which supports natively both preemptive and cooperative scheduling. This makes OpenSwarm suitable for both computationally intensive and swiftly responsive tasks.

OpenSwarm enables the developer to design platform-independent solutions. This is realised by using a hardware abstraction layer. OpenSwarm thus facilitates the implementation of high-level control strategies, and improves reproducibility of experiments across devices.

We reviewed general properties of embedded operating systems (including OpenSwarm). OpenSwarm provides a small memory footprint of 1 kB RAM and 12 kB ROM while offering a broad range of features, such as interprocess communication/synchronization.

The usage of the system was illustrated by a code example implementing a controller solution for a canonical swarm robotics problem. This code was then used in experimental trials with five physical e-puck robots. Their performance was on par with that of a hardware-near implementation from the literature.

In the future, we plan to extend OpenSwarm to provide cooperative computing across multiple devices. Furthermore, we plan to deploy OpenSwarm on further platforms—such as the Kilobot robot, which is based on Atmel's megaAVR series.

ACKNOWLEDGMENT

S.M. Trenkwalder is recipient of a DOC Fellowship of the Austrian Academy of Sciences. Y.K. Lopes acknowledges support by CAPES Foundation, Brazil (grant number: 0462/12-8). The authors gratefully acknowledge G. Kapellmann Zafra's support and thank also more than 100 students of the University of Sheffield for testing OpenSwarm.

REFERENCES

[1] E. Şahin, "Swarm robotics: From sources of inspiration to domains of application," in *Swarm Robotics*, ser. Lecture Notes in Comput. Sci. Springer, Berlin, Germany, 2005, vol. 3342, pp. 10–20.

[2] C. Bormann, M. Ersue, and A. Keranen, "Terminology for constrained-node networks," RFC 7228, 2014.

[3] M. Rubenstein, C. Ahler, and R. Nagpal, "Kilobot: A low cost scalable robot system for collective behaviors," in *Proc. 2012 IEEE Int. Conf. Robotics and Automation (ICRA)*. Piscataway, NJ: IEEE, 2012, pp. 3293–3298.

[4] S. MacConnell, *Code complete: A practical handbook of software construction*. Redmont, WA: Microsoft Press, 1993.

[5] M. O. Farooq and T. Kunz, "Operating Systems for Wireless Sensor Networks: A Survey," *Sensors*, vol. 11, no. 6, pp. 5900–5930, 2011.

[6] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating Systems for Low-End Devices in the Internet of Things: a Survey," *IEEE Internet of Things J.*, vol. PP, no. 99, pp. 1–1, 2015.

[7] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proc. 2004 IEEE Int. Conf. Local Computer Networks*. Piscataway, NJ: IEEE, 2004, pp. 455–462.

[8] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "TinyOS: An Operating System for Sensor Networks," in *Ambient Intelligence*. Springer, Berlin, Germany, 2005, pp. 115–148.

[9] F. Mondada, M. Bonani, X. Raemy, J. Pugh, C. Cianci, A. Klaptocz, S. Magnenat, J.-C. Zufferey, D. Floreano, and A. Martinoli, "The e-puck, a robot designed for education in engineering," in *Proc. 9th Conf. autonomous robot systems and competitions*, vol. 1, no. LIS-CONF-2009-004. Castelo Branco, Portugal: IPCB, 2009, pp. 59–65.

[10] M. Gauci, J. Chen, W. Li, T. J. Dodd, and R. Groß, "Clustering Objects with Robots That Do Not Compute," in *Proc. Int. Conf. Autonomous Agents and Multi-agent Systems (AAMAS)*. Richland, SC: IFAAMS, 2014, pp. 421–428.

[11] N. Mohamed, J. Al-Jaroodi, and I. Jawhar, "Middleware for Robotics: A Survey," in *Proc. 2008 IEEE Conf. Robotics, Automation and Mechatronics*. Piscataway, NJ: IEEE, 2008, pp. 736–742.

[12] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: An open-source robot operating system," in *ICRA Workshop on open source software*, vol. 3, 2009.

[13] P. Bouchier, "Embedded ROS [ROS Topics]," *IEEE Robot. Automat. Mag.*, vol. 20, no. 2, pp. 17–19, June 2013.

[14] S. Magnenat, P. Rétornaz, M. Bonani, V. Longchamp, and F. Mondada, "ASEBA: A modular architecture for event-based control of complex robots," *IEEE/ASME Trans. Mechatron.*, vol. 16, no. 2, pp. 321–329, 2011.

[15] M. Szymanski and H. Wörn, "JaMOS-A MDL2ε; based Operating System for Swarm Micro Robotics," in *Swarm Intelligence Symp. (SIS)*. Piscataway, NJ: IEEE, 2007, pp. 324–331.

[16] K. Klues, C.-J. M. Liang, J. Paek, R. Musaloiu-Elefteri, P. Levis, A. Terzis, and R. Govindan, "TOSThreads: thread-safe and non-invasive preemption in TinyOS." in *SenSys*, vol. 9, 2009, pp. 127–140.

[17] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying event-driven programming of memory-constrained embedded systems," in *Proc. 4th Int. Conf. Embedded Networked Sensor Systems*, ser. SenSys '06. New York, NY, USA: ACM, 2006, pp. 29–42.

[18] A. Dunkels, "Contiki documentation," 2015, [Online; accessed 19-July-2015]. [Online]. Available: https://github.com/contiki-os/contiki/wiki/Processes#The_Process_Scheduler

[19] W. Stallings, *Operating Systems: Internals and Design Principles*, 8th ed. Upper Saddle River, NJ: Prentice Hall Press, 2014.

[20] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He, "The LiteOS Operating System: Towards Unix-Like Abstractions for Wireless Sensor Networks," in *Proc. IPSN '08. Int. Conf. Inform. Process. in Sensor Networks*. Piscataway, NJ: IEEE, 2008, pp. 233–244.

[21] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, "MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms," *Mob. Netw. Appl.*, vol. 10, no. 4, pp. 563–579, Aug. 2005.

[22] A. Eswaran, A. Rowe, and R. Rajkumar, "Nano-RK: an energy-aware resource-centric RTOS for sensor networks," in *Proc. 26th IEEE Int. Real-Time Systems Symp. (RTSS)*. Piscataway, NJ: IEEE, 2005, pp. 10 pp.–265.

[23] G. Nutt, "NuttX - online documentation," January 2016. [Online]. Available: http://www.nuttx.org/Documentation/NuttX.html

[24] R. Hartley and F. Pipitone, "Experiments with the subsumption architecture," in *Proc. 1991 IEEE Int. Conf. Robotics and Automation (ICRA)*. Piscataway, NJ: IEEE, 1991, pp. 1652–1658 vol.2.

[25] S. M. Trenkwalder, "OpenSwarm - GitHub repository," Feburary 2016. [Online]. Available: https://github.com/OpenSwarm/OpenSwarm.git

[26] Ecole polytechnique federale de Lausanne, "e-puck library," 2014, e-puck-latest-svn-trunk.zip - 1.86 MB - 28. February 2014. [Online]. Available: http://www.e-puck.org/

[27] S. M. Trenkwalder, "OpenSwarm - online supplementary material page," February 2016. [Online]. Available: http://naturalrobotics.group.shef.ac.uk/supp/2016-001

[28] S. M. Trenkwalder, "OpenSwarm - online documentation," July 2016. [Online]. Available: http://openswarm.org/documentation

[29] A. S. Tanenbaum, *Modern operating systems*. Upper Saddle River, NJ: Prentice Hall Press, 2007.

[30] G. Sartoretti, M.-O. Hongler, M. de Oliveira, and F. Mondada, "Decentralized self-selection of swarm trajectories: From dynamical systems theory to robotic implementation," *Swarm Intelligence*, vol. 8, no. 4, pp. 329–351, 2014.

[31] V. Sperati, V. Trianni, and S. Nolfi, "Evolving coordinated group behaviours through maximisation of mean mutual information," *Swarm Intelligence*, vol. 2, no. 2-4, pp. 73–95, 2008.

[32] J. Chen, M. Gauci, W. Li, A. Kolling, and R. Groß, "Occlusion-based cooperative transport with a swarm of miniature mobile robots," *IEEE Trans. Robot.*, vol. 31, no. 2, pp. 307–321, 2015.

[33] D. Gay, P. Levis, and D. Culler, "Software design patterns for TinyOS," *SIGPLAN Not.*, vol. 40, no. 7, pp. 40–49, 2005.