VirtuosoNext: Fine-Grain Space and Time Partitioning RTOS for Distributed Heterogeneous Systems

Bernhard H.C. Sputh
Altreonic NV
Gemeentestraat 61A bus 1
Linden, Belgium
bernhard.sputh@altreonic.com

Eric Verhulst
Altreonic NV
Gemeentestraat 61A bus 1
Linden, Belgium
eric.verhulst@altreonic.com

ABSTRACT

In this paper we present VirtuosoNext, an RTOS for distributed heterogeneous systems which provides fine-grain space and time partitioning. Focusing on the Space Partitioning, the benefits and penalties of fine-grain space partitioning for real-time systems are presented by comparing VirtuosoNext to its predecessor OpenComRTOS-1.6. Both use a static memory model and can be used simultanueously in a networked system. The comparisons are based on the ARM-Cortex-M3 (MPU) and ARM-Cortex-A9 (MMU) ports of the RTOS.

General Terms

RTOS, distributed systems, space partitioning, time partioning

Keywords

RTOS, distributed systems, OpenComRTOS, VirtuosoNext

1. INTRODUCTION

A current trend in embedded systems is the use of hypervisors to allow the execution of multiple applications on the same processor. Hypervisors provide coarse-grain space and time partitioning. They provide separate memory spaces as well as separate time-slots for each application, itself composed of multiple processes or tasks. Thus it prevents the partitions interfering which each other, but does not control what happens inside a partition. In comparison VirtuosoNext, introduced in this paper, provides fine-grain Task level space partitioning, which means that individual Tasks are prevented from accessing the memory region of another Task. Hypervisors typically provide partitioned time slices usually in the order of milliseconds or tens of milliseconds. If an external event for an application arrives outside its time slice then the handling of this event stays pending until that application partition becomes active again. In VirtuosoNext we provide classical preemptive priority based scheduling of the Tasks instead of strict time partitioning. This means that external events can be handled when they occur instead of having to wait till their partition is scheduled again. The time partitioning features of VirtuosoNext are more intended as restrictions on top of the priority based scheduling, for example because safety requirements impose that a certain task should only run during a given time interval or should be terminated when it uses its allocated budget in CPU cycles.

1.1 VirtuosoNext

VirtuosoNext is based on Altreonic NV's formally developed and network-centric OpenComRTOS [3]. Like its predecessor VirtuosoNext supports the Virtual Single Processor (VSP) programming model whereby the RTOS abstracts the underlying distributed and heterogeneous processor hardware topology from the developer. This allows the developer to concentrate on the application logic without having to worry about the communication between the different Processing Nodes in the System. Furthermore, this model allows an application to be developed first on a single Processing Node and then distribute the application onto multiple Processing Nodes, without modifying the application source code. This is partly achieved by using so-called "Hubs" to provide the traditional services (synchronisation and communication between Tasks). Each service is a specific instance of a Hub. Hubs decouple the Tasks whereby Hubs and Tasks can be mapped anywhere in the system.

VirtuosoNext retained the static memory allocation at compile time that OpenComRTOS-1.6 uses. Static memory allocation by default avoids running out of memory resources during runtime by avoiding the dynamic allocation of memory. This is inherently safer than a dynamic memory allocation programming model and permits the linker to check at build time whether or not the application will fit in the available memory resources. Note that VirtuosoNext Designer, the modelling and programming environment allows the user to switch on the protection on a node by node basis with no impact on his application code.

VirtuosoNext as well as OpenComRTOS provide the following services, implemented as instances of a generic "Hub".

- Synchronisation and data exchange services provided by Hubs:
  - Port: Facilitates a synchronised exchange of data between two Tasks, similar to a CSP-Channel.

  - Event: Represents a boolean event, which can be raised and signalled. While it is in the raised state any further request to raise it is put on the waiting list of the Hub, if the interaction semantics permit it (_W and _WT).
  - Semaphore: Represents a counting event, which can be signalled and tested. This means that it can be raised multiple times by the same Task until a user defined maximum value has been reached, upon which any further requests are enqueued on the waiting list.
  - Fifo: Buffered data exhange between Tasks. Works system wide.
  - Memory Block Queue (MBQ): This Hub provides the ability to exchange blocks of memory between Tasks. Tasks can allocate these memory-blocks from the MBQ-Hub and then fill them with the desired data. The data exchange is a zero copy operation (on the same Node), as just pointers to the memory-blocks get exchanged. The size of the memory blocks is user definable.
  - BlackBoard: This is a safe system wide global data structure. All read and write operations to this data structure are performed atomically.
  - DataEvent: Combines an event with a message. When the DataEvent gets signalled the signalling Task can put a message for the Task that tests the DataEvent. Once the DataEvent has been tested the message is purged from it. A DataEvent can be raised multiple times, before being tested, to update the message stored in it.
  - Resource: Represents a lock which provides system wide (read: distributed) priority inheritance.
  - MemoryPool: The MemoryPool allows a Task to allocate blocks of memory for local use. This Hub only works locally, i.e. Task and Hub must ben on the same Node.
  - PacketPool: Allows a Task to allocate additional L1_Packets to use for asynchronous interactions. Like the MBQ- and MemoryPool-Hubs this Hub only works locally.

- Task State Manipulation: VirtuosoNext allows to suspend and resume Tasks. It is not possible to create a new Task during runtime, as this would violate the static allocation scheme used by VirtuosoNext. However, the developer can use a pool of Tasks, created at compile time.

- Timer Services: Tasks can wait and deschedule for a time interval, as well as their interactions may time out (more in the next paragraph).

Interactions can have the following interaction semantics:

- _W (Waiting): The Interaction will block until synchronisation is achieved, before returning to the Task that attempted the interaction. In case the interaction succeeded the return value RC_OK is provided. Should the interaction fail, due to not being permitted then RC_FAIL will be returned immediately.

- _WT (Waiting with Timeout): The interaction will block until either synchronisation is achieved or the timeout expired, before returning to the Task that attempted the interaction. In case the timeout expired an interaction will return RC_TO to the Task.

- _NW (Non Waiting): The interaction will test whether or not synchronisation has been achieved and then immediately return to the Task that attempted the interaction. In case the interaction failed the the return value RC_FAIL is provided.

- _A (Asynchronous): An Asynchronous Interaction is an Interaction whereby the issuing Task can continue while the Interaction is being processed. A Task may have multiple Asynchronous Interactions pending at the same time. Internally Asynchronous Interactions are treated like Waiting ones. Once the Task has issued all the Asynchronous requests it has to wait for them to succeed. This is also why Asynchronous Interactions are also called two-phase services. Async-Services are useful when a Task has to wait for one of multiple possible choices to happen, for instance for an interrupt from a hardware device and at the same time for a request from another Task.

1.2 Space Partitioning

Fine-grain Task level space partitioning compared to process or application level space partitioning has the advantage that it allows a much finer level of partitioning which can be as small as a single line of code, but typically the function providing the Task entry point, without jeopardising the real-time capability, an issue that traditional hypervisor type approaches have. In process level space partitioning the data of individual threads of a process are shared, which means they can corrupt each other's data. This is not the case when using the space partitioning support of VirtuosoNext, whereby each Task runs in User-Mode and is only permitted to access its own memory (allocated at compile time), as well as explicitly shared memory in the form of global variables. This also prevents direct access to the underlying hardware for which the application Task can call the trusted services of the underlying RTOS kernel and its driver layer. As in VirtuosoNext drivers are implemented as Tasks, the user can also develop them in Supervisor-mode.

With VirtuosoNext the application is now split explicitly between a trusted and a non-trusted zone. The trusted zone contains the qualified kernel Task and the driver layers. The untrusted zone contains the application Tasks that can use the services provided by the trusted zone. In this case the kernel Task can be fully trusted as it underwent a qualification process.

1.3 Time Partitioning

VirtuosoNext does not provide a classical time partitioning implementation as seen in hypervisors, instead like its predecessor OpenComRTOS it provides system wide (distributed) priority based preemptive scheduling at all levels. This means that a high priority request from Task A on Node 1 for a Service provided at Node 23 will be treated everywhere as a high priority request. This means that with the exception of some memory and scheduling overhead, VirtuosoNext provides the responsiveness of a traditional RTOS (like OpenComRTOS), albeit in a distributed implementation and classical scheduling theories remain valid. Rather than allocation fixed and isolating time slots (that put a serious lower boundary on the reaction time of the system), VirtuosoNext provides support for restricting the scheduling in time of Tasks on top of the priority based scheduling. For example, Tasks can be defined with earliest starting time and latest termination times or with a maximum CPU cycles budget. The kernel Task continuously monitors these tasks specific boundary conditions. Note however, that such boundary conditions are often not needed, unless safety requirements impose them.

1.4 Outline

In the following Section **??** we first give an introduction into VirtuosoNext's space partitioning concepts. It covers the implementation differences of space partitioning between an MPU and an MMU, based on the implementation for an ARM-Cortex-M3 [2] and an ARM-Cortex-A9 [1]. Besides the Kernel-level implementation it also includes the support tools and the build process. The impact of the space partitioning implementation onto the code size and the runtime behaviour are examined in Section 5, by comparing VirtuosoNext to its predecessor OpenComRTOS-1.6. The results are discussed in Section 6. This is followed by Conclusions and Further Work in Section 7.

2. IMPLEMENTATION

To perform task-level fine-grain space partitioning the RTOS requires hardware support, either a Memory Protection Unit (MPU) or a Memory Management Unit (MMU). These units allow one to assign access properties to the different memory regions. Typical properties are:

- Read-Only: The memory region may only be read from, but not written to. This is used in VirtuosoNext for all the regions that contain instructions.
- Read-Write: The memory region that can be read from and written to. This applies to the data region a Task may access.
- Non-Executable: The memory region that may not be used to fetch instructions from. If supported by an MPU/MMU unit then VirtuosoNext utilises this property for all memory regions that do not contain instructions.

In VirtuosoNext the following memory regions are defined:

- Shared regions:
  - Code-Shared: Contains the instructions for the whole Node. If possible this region is marked read-only and kept in the Flash memory of the SoC.
  - Data-Shared: Contains data that is shared among all Tasks on the Node. For instance, the Packets of a Packet-Pool-Hub (managed by the trusted Kernel task).
  - BSS-Shared: Contains zero initialised data for all Tasks of the Node. For instance the trace buffer when enabled.
- Task specific regions:
  - Data-Task-N: Data that is specific to a Task, this includes, among other things, the Request-Packet, the Task Context and the Task Control Block. For the Kernel Task this contains also the definition of the local Hubs.
  - BSS-Task-N: Zero initialised data of the Task, such as the stack of the Task.

2.1 Privilege levels

VirtuosoNext provides two privilege levels:

- User-Level: A Task at this level may only access its own private data, the data shared in the Data-Shared and BSS-Shared regions, and execute the instructions that are in the Code-Shared region.
- Supervisor-Level: A Task at this level, may access the whole memory available without any restrictions or monitoring. This level is used by the Kernel-Task, Interrupt Service Routines, and for device drivers which need to directly interface to their memory mapped devices. These elements form the trusted zone, and can only be accessed from User-Level Tasks through guarded interfaces that ensure that no malicious request passes through.

2.2 Adjustments to the RTOS Implementation

In order to support memory protection the VirtuosoNext implementation was changed in the following areas:

- Data Structures: The context of a Task was changed to contain information about the private memory regions of the Task, as well as the privilege level of the Task.
- Boot procedure: Configures the shared memory regions in the MPU before enabling the MPU.
- Context switch: The context switch must reconfigure the MPU with the Task private memory-regions. Furthermore, the context switch must now be executed in the supervisor mode of the CPU to allow reconfiguration of the MPU / MMU.
- User-Task to Kernel-Task communication had to be adjusted to allow the User-Task access to the Kernel-Task interface.

2.3 Illegal Accesses

When a Task tries to access a memory location in a way that is not permitted the MPU/MMU triggers a CPU-exception. In VirtuosoNext this results in a transfer to the Task's Abort-Handler followed by restarting the Task.

3. IMPLEMENTATION DIFFERENCES BETWEEN MPU AND MMU

The hardware support for memory protection is usually done by either a Memory Protection Unit (MPU) or an Memory Management Unit (MMU). VirtuosoNext supports the ARM-Cortex-M3 MPU as well as the ARM-Cortex-A9 MMU although this can be ported to other processors as well. This section details the implementation differences between the ARM MPU and MMU.

3.1 ARM-Cortex-M3 (MPU)

The MPU of the ARM-Cortex-M3 SoC [2] (also used by the ARM-Cortex-M4 and the small ARM-Cortex-R4 SoCs) allows to mark a memory region as RO, RW, and Non-Executable. This MPU is designed for small systems and allows to only have up to eight regions for a currently running Task in parallel. Furthermore, it has strict rules on the alignment and size of a memory region. A memory region can only have sizes that are 2^n with $n \in \{\}$ with a minimum size of 4kB and the alignment of the region is the same as the size of it. Thus the starting address of a 16kB region must be aligned to 16kB. In addition this MPU allows to split a memory region into 8 sub-regions, although this was not used in the implementation.

Due to the previously mentioned constraints it is clear that this MPU is meant to be used in systems where the developer manually finetunes the linker script. This is undesirable as it is an error prone and time intensive process and thus we decided to automate it. Automation of MPU usage faces two challenges:

1. Generate a correct linker script to correctly align the memory regions, for which one needs to know the sizes of the different memory regions, to properly align them.
2. Determining the sizes of the different memory regions. Before building an application one does not know the size of the different memory regions.

VirtuosoNext overcomes these challenges by using an adjusted build procedure for ARM-Cortex-M3 Nodes, which is also applicable to the M4 and some instances of the R4. The procedure is as follows:

1. Generate all the files for the ARM-Cortex-M3 Node, and assign the data structures to their corresponding memory regions. It is important to note that the linker script generated in this step contains all the necessary memory regions and stores an integer multiple of their placement in predefined symbols in the program.
2. Build the Node.
3. Generate a map file for the Node allowing to know the sizes of the different memory regions.
4. Generate an adjusted linker script that properly aligns the memory-regions, based on the previously generated map file. In this step VirtuosoNext uses a new tool called the SectionAnalyser-Arm-Cortex-M, which calculates the alignment and potential necessary paddings.
5. Build the Node using the adjusted linker script, generated in the previous step.

The previously outlined build process allows the developer to utilise the ARM-Cortex-M3 MPU without having to manually configure it. Hence, the memory partitioning and protection is transparent for the application code itself.

3.2 ARM-Cortex-A9 (MMU)

The ARM-Cortex-A9 MMU [1] is more flexible than the MPU of the ARM-Cortex-M3. It does not impose a limitation between size of a region and its alignment. It operates with a granularity of 4kB, which means that a page in the MMU always has the size of 4kB. However, this comes not for free, the developer has to provide the MMU with information about each page a Task is allowed to access, in the form of a Table. Updating this Table makes the context switch quite complex and thus expensive in cycles, and results in a larger memory footprint of the final binary. See the Results section for detailed figures.

4. TIME PARTITIONING

VirtuosoNext provides a priority based preemptive scheduling model. Each Task in the system has assigned a priority to it and the scheduler always executes the Task with the highest priority. This applies global scheduling across all Tasks and hence all applications. To achieve time partitioning in the hypervisor sense this can be achieved by allocating a range of Priorities to every Partition, and then use a ticker-timer for each Partition which will raise an Event when the time-slot for the partition has been reached. Using this Event the Tasks of the Partition activate and then perform their work as if they were scheduled in tim. This model works single processor as well as in a distributed system, if low latency links between the different processing Nodes are used. Typical hypervisor solutions cannot be used in distributed heterogeneous systems by default, and they are usually also specific to the used hardware. This is not the case with VirtuosoNext.

To react to external events on time an Application may register Interrupt Service Routines (depending on the used hardware and the device driver). Interrupt Service Routines have priority over any application Task. Interrupt Service Routines may interact with Tasks using VirtuosoNext-Hubs using Non-Waiting Semantics, allowing them to activate Tasks upon external events and using an Application-Task to handle them. These ISR-Event handling Application Tasks will be placed in the highest Priority-Group, higher than any Partition, which allows them to run immediately after an external event occurred. These Tasks are usually not computation expensive, they might just read some data from a Device and make some control decisions, write some information to it or to another device, or to pass the data to another Application-Task. Thus control is quickly returned to the active Priority Group. This scheme allows VirtuosoNext to serve external events as quickly as possible, while at the same time it permits execution of individual Applications to happen only in predefined time slots.

VirtuosoNext does not prevent that a partition is not overrunning its preallocated time slot because such a time slot is virtual. If this is desired the user can however manually suspend a group of Tasks, to prevent them from getting any more runtime. However, in most embedded applications a slight overrunning partition is not a major problem, if it does not happen continuously. In this case the partition time slot has been specified too short or it is an indication of a hardware issue.

Form above, it is clear that a better scheme is to use global priority based scheduling for all Tasks across all Application Partitions. Often a Rate Monotonic Analysis will provide a good starting point to assign the priorities as most applications tasks are periodic. However, if safety requirements impose it, Tasks can be specified with restrictions on when they can start and must terminate and how many cycles they are allowed to consume per period. This is not time partitioning in the strict sense of the word but a way to exercise fine grain control over the execution of application tasks.

5. RESULTS

5.1 Code Size

The fine-grain space partitioning implementation of VirtuosoNext is lightweight both in code size and in runtime impact. The code size of OpenComRTOS-1.6 (used as a reference) and the VirtuosoNext implementation was compared by building the same application using all available Services (compiled with Os). For the ARM-Cortex-M3 platform (TI-LM3S6965 SoC [5]) the code size increased by 2908 bytes to 11564 bytes. For the ARM-Cortex-A9 platform (TI-OMAP4460 [4]) the code size increased by 6700 bytes to 21844 bytes. The larger increase is due to the more complex configuration needed for the MMU of the ARM-Cortex-A9 compared to the MPU used by the ARM-Cortex-M3.

5.2 Semaphore Loop Times

Space partitioning also affects runtime performance because the context of a Task now includes also the information about the memory regions it is allowed to access. This becomes visible when comparing the time the system takes to perform a Semaphore-Loop (two Tasks, two semaphore Hubs with one loop requiring eight context switches). For the ARM-Cortex-M3 (@50MHz) the execution time per loop has increased from 2730 to 2945 clock cycles (compiled at O3) and for the ARM-Cortex-A9 (@700MHz) the time increased from 16557 to 21271 clock cycles (compiled at O3).

5.3 Interrupt Handling Latency

In addition to fast context switching a RTOS must also be able to react predictably and with very low latency to external events, so called Interrupts. In the case of VirtuososNext and OpenComRTOS we define two Interrupt Latencies of interest. The first one is the Interrupt Request (IRQ) to ISR (Interrupt Service Routine) Latency, the second is the IRQ to Task Latency. For the ARM-Cortex-M3 (@50MHz) the minimal IRQ to ISR Latency increased from 46 to 50 clock cycles (compiled with Os) and the IRQ to Task Latency increased from 754 to 850 clock (compiled with Os). The impact on the ARM-Cortex-A9 (@700MHz) is that the minimal IRQ to ISR Latency increased from 800 to 850 clock cycles (compiled with O3) and the IRQ to Task Latency from 1420 to 2465 clock cycles (compiled with O3). Note that the interrupt latency is really a histogram as it depends on what other applications are active on the processing node. To simulate such a stress pattern, the above mentioned semaphore loop is scheduled in parallel with the interrupt agency measurement. The semaphore loop is a very good stress load as it continuously disables interrupts for short interval when the Kernel Task execute the semaphore services and executes context switches.

5.4 Virtual Time Partitioning

To illustrate the feasibility of the time partitioning support of VirtuosoNext we implemented a system, consisting of one Node (thus a single CPU) that executes three independent and periodic applications (100ms periodicity): a controller for a skid steered vehicle (Tasks with the IDs: 2, – 8; and Hub IDs: 0–3 and 7), a simulation of an inverted pendulum monocycle (Tasks with the IDs: 9 – 12; Hub IDs: 8 –12), and a SemaphoreLoop burst (Task IDs: 13, 14; Hub IDs: 13, 14). Figure 1 is an Event Trace of the application with all Tasks executing at the same priority. In both cases is the order of execution first determined by the priority and next by the partial order imposed by the interactions. Not that in a real example, the priorities within an Application would also differ and be determined by their periodicity. An Event Trace is a recording of the scheduling events and the interactions with the Hubs, which can be graphically displayed using the Event Tracer tool provided by VirtuosoNext. In the Event Trace of Figure 1 all the application Tasks are running at the same priority of 128, we can see that each application Task gets scheduled when it is at the top of the ready list. Furthermore, we see at the end of the trace the scheduling of the Idle-Task (Task ID: 2), which indicates that currently there is no Task that is ready. In Figure 2 the three applications have been scheduled with distinct Priorities, with the skid steering controller application having the highest Priority (64), the inverted pendulum having a medium Priority (128), and the SemaphoreLoop having the lowest Priority (128). We can see how first the skid steering controller application executes first, followed by the inverted pendulum, which then is followed by the SemaphoreLoop. Furthermore, we again see at the end of the graph the Idle-Task being scheduled, followed by the skid steering controller application. This illustrates that it is possible to achieve time and space partitioning using the current priority based preemptive scheduling without loosing much of the reactivity of a classical RTOS.

```
Must find the example and get:  cycle time and period
length from them, the priorities.
```

6. DISCUSSION

The results presented in the previous section show that using an MMU compared to an MPU has a larger impact on both code size and runtime, without having a general benefit in our standard benchmarks. The code size of the ARM-Cortex-A9 kernel increased by 33.1% while the code size of the ARM-Cortex-M3 increased only by 1.9%. From past experience we know that a larger code size usually results in slower runtimes, due to code having a lower probability of not fitting into the instruction cache.

The SemaphoreLoop time on the ARM-Cortex-A9 increased by 28.5% while for ARM-Cortex-M3 it only increased by

7.9%, this directly reflects the much larger context switch on the ARM-Cortex-A9 due to the additional code to handle the MMU compared to the few additional registers for the MPU of the ARM-Cortex-M3. A very large increase is also visible for the IRQ to ISR latency which increased by 38% on the ARM-Cortex-A9, while at the same time there was no change on the ARM-Cortex-M3. The IRQ to Task latency increased by 73.6% for the ARM-Cortex-A9, which is almost three times more than the increase of the SemaphoreLoop time. For the ARM-Cortex-M3 it only increased by 6.3% which is less than the increase in the SemaphoreLoop time. This indicates that there is most likely an issue with code that handles the MMU inside the interrupt handling code used by the ARM-Cortex-A9.

7. CONCLUSIONS AND FURTHER WORK

This paper introduced VirtuosoNext with a focus on achieving fine-grain and time partitioning to preserve the reactivity of a static RTOS and gain the benefits of protection of on-chip memory protection. Two space partitioning implementations were introduced, the MPU approach as used by the ARM-Cortex-M3, and other MCUs of the same league, and the MMU approach as used by the ARM-Cortex-A9 and similar processors. It was shown that both approaches work very well. The difficulties of utilising them where highlighted as well as how they have been overcome in VirtuosoNext. Furthermore, the VirtuososNext implemnation was compared to its predecessor OpenComRTOS-1.6, on which the development of VirtuosoNext was based. The comparison was both on code size as well as runtime performance, and it showed that an MPU results in a smaller performance impact than an MMU, and that we should take a look at the ARM-Cortex-A9 implementation to see whether or not we can optimise it further.

This brings us to the future work. As already mentioned the ARM-Cortex-A9 implementation can be further optimised. There is also a potential to improve the ARM-Cortex-M3 MPU handling by using the sub-regions offered as this would reduce the size of the memory regions we have to protect. This would require to improve the Section Analyser tool. Naturally, we also would like to expand our space partitioning support also to other CPU families, such as Freescale PowerPC, Synopsys ARC.

8. ACKNOWLEDGMENTS

The authors would like to thank Antonio Ramos for implementing the MMU support for the ARM-Cortex-A9.

9. REFERENCES

[1] ARM. *ARM Cortex-A9 Processor Technical Reference Manual*, revision r4p1 edition, 2012.

[2] ARM. *ARM Cortex-M3 Processor Technical Reference Manual*, revision r2p1 edition, 2015.

[3] B. H. Sputh, E. Verhulst, and V. Mezhuyev. OpenComRTOS: Formally developed RTOS for Heterogeneous Systems. In *Embedded World Conference 2010*, Mar. 2010.

[4] Texas Instruments. *OMAP4460 Multimedia Device Silicon Revision 1.x Texas Instruments OMAP Family of Products Version AB Technical Reference Manual*, 2014.

[5] Texas Instruments. *Stellaris LM3S6965 Microcontroller*, 2014.

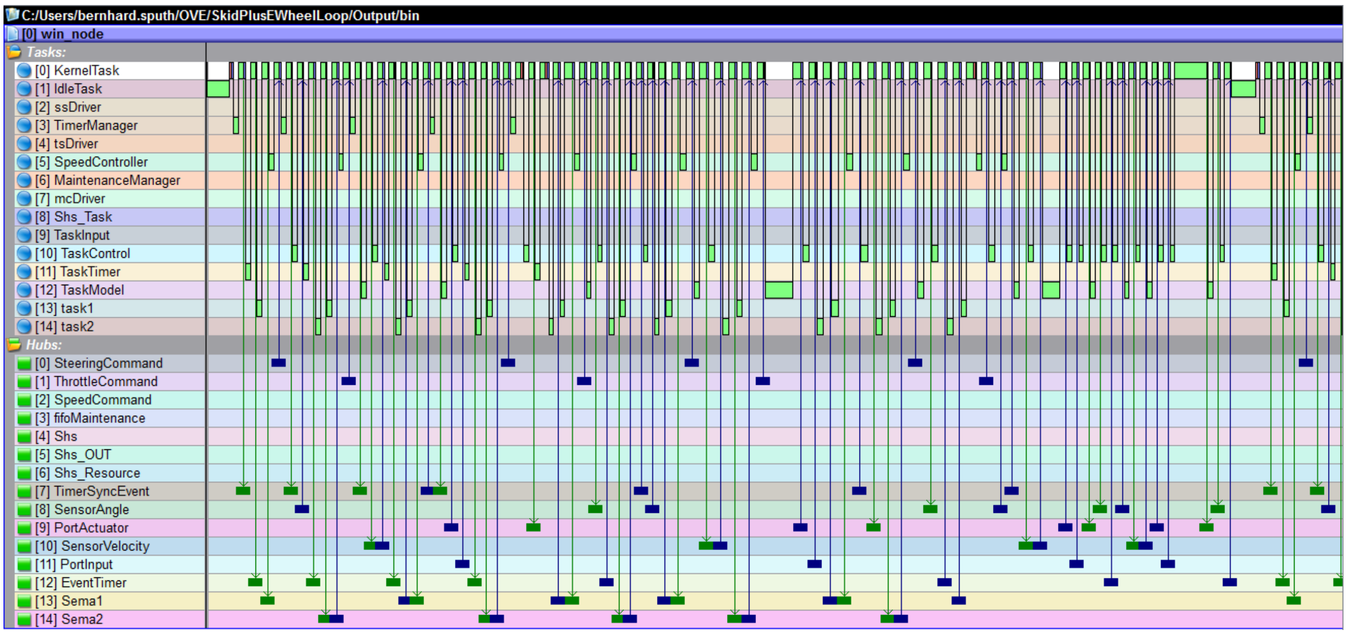

Figure 1: Three differrent applications running at the same Priority

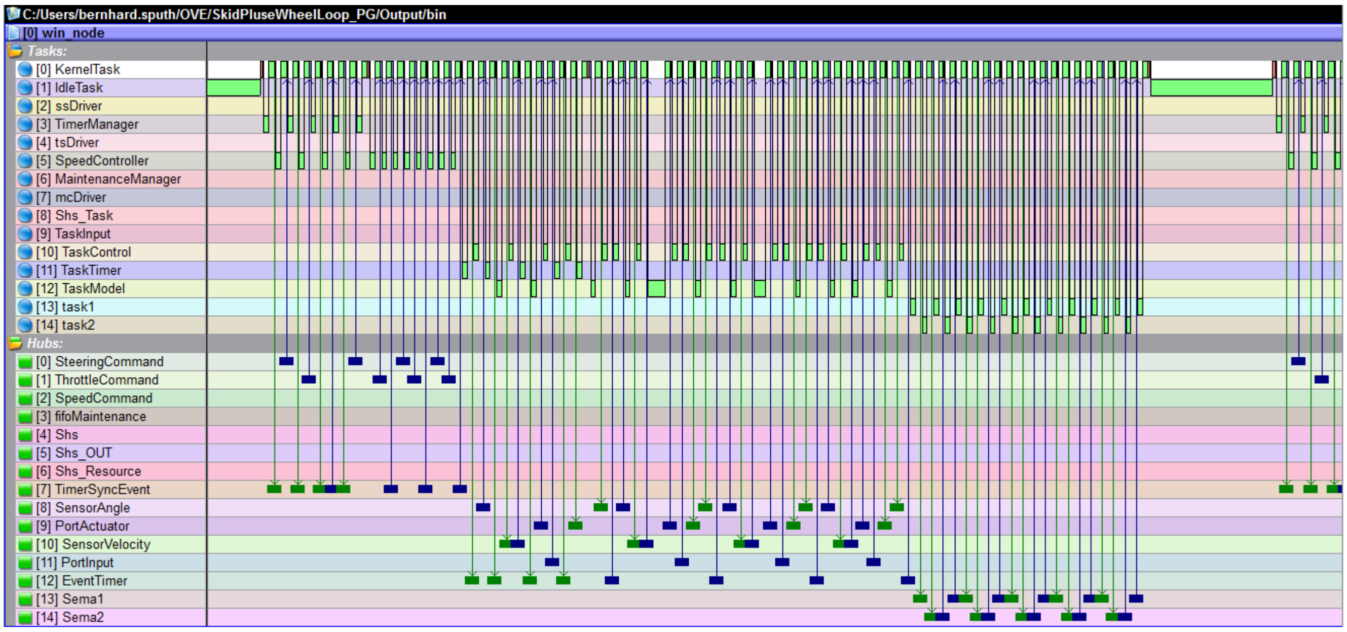

Figure 2: Three differrent applications running at distinct Priorities