Design and Implementation of an EDF Scheduler for Fiasco.OC L4

Eric Arnerlöv
Abstract

Design and Implementation of an EDF Scheduler for Fiasco.OC L4

Eric Arnerlöv

This paper describes the implementation of an EDF scheduler for the Fiasco microkernel. Benchmark test shows that the average case gain an improvement of 16-34%, in regards to schedule calls, running EDF over the existing fixed priority scheduler using Rate Monotonic (RM). A possible approach to mimic EDF scheduling with fixed priority is described using dynamic counters and worst case ready queue. The advantages and disadvantages are compared to the Fixed Priority with the Priority Promotion algorithm.
Contents

1 Introduction 4

2 Background 4
   2.1 Real-Time Systems ........................................... 4
   2.2 Real-Time Task ............................................ 4
   2.3 Task Activation Modeling .................................. 5
   2.4 Preemption .................................................. 5
      2.4.1 Priority Inversion ..................................... 5
      2.4.2 Priority Inheritance .................................. 6
   2.5 Scheduling Algorithms ..................................... 6
      2.5.1 Fixed Priority ......................................... 6
      2.5.2 Rate Monotonic ........................................ 6
      2.5.3 Earliest Deadline First ............................... 6
   2.6 Fiasco.OC .................................................. 7
   2.7 Threads and Queuing in FIASCO ....................... 7
      2.7.1 Ready Queue ........................................... 8
      2.7.2 Scheduling Process and Lazy Queuing .............. 9
   2.8 Timeouts .................................................... 10
   2.9 Inter-process Communication ............................... 11
      2.9.1 Thread Synchronization using IPC ................ 11

3 Implementation 12
   3.1 Additional Timeouts ..................................... 14
   3.2 Changed Made to the Schedule Context Class ........ 14
      3.2.1 Additional Variables .................................. 14
      3.2.2 dominates(...) ....................................... 15
      3.2.3 in_ready_list() ....................................... 15
      3.2.4 setup realtime_thread(...) .......................... 15
      3.2.5 change_thread_param(...) ........................... 16
      3.2.6 set(...) ............................................... 17
      3.2.7 Destructor ............................................. 17
   3.3 Changes to the Ready Queue ............................... 17
      3.3.1 System and Real-time Mode ......................... 17
      3.3.2 Additional Variables .................................. 17
      3.3.3 set_scheduling realtime_mode(....) ................ 18
      3.3.4 Enqueue and Dequeue ................................ 18
   3.4 User Calls to the Kernel ................................ 19
      3.4.1 l4 scheduler run thread edf(...) .................. 19
      3.4.2 l4 scheduler work done(...) ........................ 19
      3.4.3 l4 scheduler set thread param(...) ................ 19
      3.4.4 l4 scheduler release thread(...) ................. 20
      3.4.5 l4 scheduler set realtime mode(...) ............... 20
   3.5 Create a Sporadic Task with Accumulated Bursts ...... 20

2
4 Testing the Implementation ........................................... 22
  4.1 Timing ................................................................ 22
  4.2 EDF Queue ......................................................... 22
  4.3 Schedule Calls .................................................... 22
  4.4 IPC Sender Queue ............................................... 24
  4.5 Aperiodic Releases ............................................. 25
  4.6 Basic Pthreads ................................................... 26

5 Benchmark: EDF vs RM ............................................. 27
  5.1 Benchmark Result ................................................. 29

6 Alternative Implementation of EDF ................................. 32
  6.1 Fixed Priority with Priority Promotions ....................... 32
  6.2 Mimic EDF with Fixed Priority using Dynamic Counters and
      Worst Case Ready Queue ........................................... 33
      6.2.1 Dynamic Counters and WCRQ ............................... 33
      6.2.2 An Approach to Mimic EDF using Fixed Priority ....... 35

7 Discussion .................................................................. 36

8 Conclusion ................................................................ 38

9 Future Work ................................................................ 39

10 Known Implementation Issues .................................... 39

List of Figures

1 Example of Priority Inversion ........................................ 6
2 An illustration of the Ready Queue ................................ 8
3 Example of removing node 4 from the heap in O(log n) time. 13
4 Improvements in schedule calls .................................... 29
5 Improvements in enqueue/dequeue Calls ......................... 30
6 Improvements in enqueue time for RM ............................ 30
7 Improvements in enqueue time for EDF ........................... 31
8 Improvements in dequeue time for RM ......................... 31
9 Improvements in dequeue time for EDF .......................... 32
10 An illustration of the smallest absolute deadline difference. 36
11 Comparison between EDF and RM in regards to dequeue time. 37
1 Introduction

Not many microkernels provide the Earliest Deadline First (EDF) policy in the kernel space. Having EDF implemented in user-space above fixed priority scheduling comes with problems such as mapping priorities to absolute deadlines. Therefore, an implementation of EDF is made in the kernel space in the FIASCO microkernel that easy allows extensions to be made in user-space. Since there have been many discussions about the overhead of Earliest Deadline First (EDF) compared to RM, a benchmark for EDF and RM and is performed on the FIASCO microkernel. The result of the benchmarks is discussed and conclusions are made in regards to schedule calls and time spent in the ready queue. An idea to combine fixed priority with EDF is made and advantages as well as disadvantages between the already existing solution Fixed Priority with Priority Promotions (FPP) and the proposed solution is discussed.

2 Background

2.1 Real-Time Systems

In our everyday life we are around systems which are dependent on timing constraints. An example of such system is the Anti-Lock Braking System (ABS) in a car. The system prevents the driver from pushing the brakes too hard which could lead to skidding. This works by having a sensor monitoring the rotation speed of a wheel when the driver pushes the brakes and keeps it from a total stop by letting a series of hydraulic valves limit the brake pressure on the wheels when it detects a possible full stop. It is critical that the sensor repeatedly reads the rotation speed and deliver the current information to the hydraulic control system within a certain time frame. A sensor delay or failure could lead to a full stop of the wheel and cause skidding or worse, decreasing the braking pressure when it is not needed causing a longer braking distance for the car. This system is an example of a real-time system. A real-time system is a system which have one or more time-critical tasks which it is critical for that task to respond within a certain amount of time for the whole system to function correctly.

2.2 Real-Time Task

A real-time task with timing constraint can be defined by its release time, activation time, completion time and deadline. A released instance of a task is referred to as a job. The release time is the time point of which the job becomes ready to execute. Activation time is the time point when the job first starts to execute and the completion time is the time point when the job is done with its execution. When the completion time of a job exceeds a certain time point for which it starts to effect the system in a negative way, it is referred to as a deadline. There are two types of deadlines, soft and hard. If the task execution time exceeds a soft deadline, it will lower the quality of the system. If the task exceeds a hard deadline, it might result in a system failure.

There are different release behaviours of a real-time task. A task may be periodic which means that it will be released at a fixed frequency depending on
its assigned period, e.g. sensor readings. A strict periodic task is a task with a relative deadline equal to its period. If the release pattern is unknown in advance, e.g. when the user of the system pressed a button, it is called an aperiodic task. An aperiodic task with a hard deadline as well as a known minimum inter-arrival time between two job releases of the same task is called sporadic. A sporadic task may also be released multiple times at once. However, between each bursts the minimum inter-arrival time must still be respected. This burst behaviour is referred to as sporadic task with burst capability.

2.3 Task Activation Modeling

There exists different models to describe the activations of a task. For example, the PJD-model describes periodical job releases with jitter and a minimum release distance. Another example is Real-Time Calculus which can be used to derive an upper and lower bounding curve of job releases. Both models features synchronous job releases of a task and allows only a single job release per task and instant of time. Because of the generic way RTC describes job releases with bounding curves, different activation patterns, for example described by the PJD-model, can be transformed into RTC curves. The transformation from PJD into RTC curves is discussed in [10].

2.4 Preemption

It is important for a real-time system to be responsive and to be able to quickly handle external and internal interrupts. For example, an external interrupt could be a sensor detecting an abnormal behaviour which need to be taken care of as soon as possible to avoid damage to the system. Allowing preemptions can help to increase the responsiveness of a system by allowing jobs to be temporarily interrupted by other jobs with higher priority. When the higher priority job has finished its execution, the preempted job will resume its execution where it received the interrupt. The process of switching jobs requires the state of the current executing job to be saved and the state of the higher priority job to be loaded.

2.4.1 Priority Inversion

One problem when dealing with synchronization is that a higher priority jobs can be blocked by a lower priority task. The situation is called Priority Inversion and it can slow down the system responsiveness and increase the chance of a deadline miss. An example of Priority Inversion is the scenario where there exists three tasks H, M and L and also a shared resource R. Each task has a different priority assigned to them. H got the highest, M has medium and L has the lowest priority. The scenario starts by an activation of task L. During its execution, L acquires the shared resource R. The job of task L then get preempted by the release of the higher priority task H. Job H tries to acquire the resource R but becomes blocked because R is already being held by L. After some time, M get released and because of its higher priority, M will preempt L. This leads to a situation where the high priority task H will be blocked by a lower priority task and the priority arrangements is violated.
2.4.2 Priority Inheritance

Different methods to prevent Priority Inversion have been created. One method is called Priority Inheritance, which is also used in the microkernel FIASCO. It works by temporarily increasing the priority of a job A, which holds a shared resource R, to the highest priority of the jobs waiting to lock R. Once job A releases the resource, its priority is downgraded to its original assigned priority. By increasing the priority of the job holding the resource, you avoid the problem of Priority Inversion and assure progress towards releasing the locked resource while still holding the priority arrangements.

2.5 Scheduling Algorithms

The purpose of the scheduler is to decide which, and for how long an instance of a task should be executed on the CPU. The choice of scheduling algorithm affects the systems features such as responsiveness and CPU utilization.

2.5.1 Fixed Priority

A common static scheduler is the fixed priority scheduling. It is popular because of its simplicity when it comes to implementation. Each task is assigned a fixed priority in advance and the scheduler will choose to execute the job with the highest priority. If there are several ready jobs with the same priority, there are different solutions for distributing the CPU time among them. For example, FIASCO will schedule them in a Round Robin fashion.

2.5.2 Rate Monotonic

A fixed priority scheduler often used in real-time systems is Rate Monotonic(RM). The priorities are statically assigned to threads in regards to their period and shorter periods means higher priority. It has been proven that RM guarantees that all deadlines are met for n tasks if the CPU utilization is below or equal to $n \times (2^{1/n} - 1)$ [8].

2.5.3 Earliest Deadline First

Earliest Deadline First (EDF) is an optimal scheduling algorithm. An optimal scheduler can also schedule a set of tasks $T$, such that all deadlines are met, if there exist any scheduler that can schedule $T$. EDF dynamically assigns the priority of a task in regards to its absolute deadline. The job with the nearest
absolute deadline will have the highest priority and the job with the longest absolute deadline will have the lowest priority. The EDF scheduler is proven feasible for a set of strictly periodical tasks when $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$ where $C_i$ is the worst execution time and $T_i$ is the period for a task $i$ [8].

2.6 Fiasco.OC

Fiasco.OC is a 3rd-generation microkernel based on the L4 microkernel. The OC refers to "Object Capability System" which is the idea that the system is based on objects which can be called to provide services, called servers. Each server can be called only if the client, an object that request a service from a server, has the capability of the server. A capability can be seen as the address to an object. For a client to communicate with a server, it is necessarily for that client to have the capability of the destination server to be able to initiate a communication between them.

The main idea behind a microkernel is that it should only contain the necessarily core functionality, such as IPC calls, basic scheduling of tasks and basic memory management. Other system services such as a virtual file system and IO management should be implemented in user space. This allows operating-systems to be built on top of the microkernels more easily and provide flexibility and modulation. Microkernels also facilitate verification of the kernel itself due to the small size and complexity. Another benefit of having system servers as separated processes in user-space is that an error to one server might not necessarily bring down the whole system. It might just be enough to restart the faulting server instead of the whole system. The downside of having isolated servers is the increased communication overhead, which was a serious problem with the first generation of microkernels, such as Mach [2], but was greatly improved in L4.

2.7 Threads and Queuing in FIASCO

FIASCO separates the executable and the scheduling parameters for a thread into two classes. A class named Context which is the actual executable thread and Sched_context that contains all the parameters needed to schedule a thread such as priority and time quantum. This standard scheduling context for a specific thread is always accessible by sched_context() which is located in the context class.

Whenever a thread is created, it will be associated with one original scheduling context that is stored in the thread control block (TCB). Each thread has one TCB and an extension called User-level Thread Control Block (UTCB). The UTCB contains data from the user space and are mostly used to store system call parameters.

A context may not always be using its standard scheduling context. A thread can donate its scheduling context to another thread and therefore let it run with different scheduling parameters. Example of such donation would be to prevent priority inversion during an IPC call where a thread requests a service from a lower priority one. It is also common to donate scheduling contexts during the use of locked resources. If a thread needs to access a shared resource which is taken by a lower priority one, the higher priority can help the lower priority thread to release its resource faster by donating its scheduling context.
The function `sched()` is used to access the current scheduling context used, not necessarily its original in case of donation, by a thread. The ready list always keeps a reference to the current executing scheduling context chosen by the scheduler. This reference is accessible by `current_sched()`. The `context()` will return the thread that was originally associated with a scheduling context and the current executing thread in the system can be accessed by `current()`.

Each thread in FASCO has several state values which can be found in the `thread_state` class. By examining the states of a thread one can for example tell if a thread is engaged in a IPC call and what phase of the IPC it is currently in. One very important state is the Thread_ready state. It indicates if a thread is ready to be scheduled or not.

In the case of a multi-core system, each CPU core is isolated from the others, however, it is possible to migrate threads between cores. Each core gets its own local ready queue and if a scheduling decision is to be made, it will be done locally among the threads belonging to that core.

### 2.7.1 Ready Queue

The standard scheduling algorithm used in FIASCO is fixed priority. The priorities are represented as an 8-bit unsigned integer that gives a priority range from 0 to 255 where higher value means higher priority. Each CPU core has its own local ready queue for its local threads that consist of an array with 256 entries. Each entry represents a priority and contains a circular linked list. The circular linked lists are made by linking scheduling context. The schedule context inherits a class named `D_list_item` which will add the basic functionality of a linked list such as accessing the next and previous element.

![Figure 2: An illustration of the Ready Queue.](image)

The ready queue also keeps tracks of the highest priority currently existing in the queue as a variable named "prio_highest". Whenever a enqueued or
dequeued operation is requested, the ready queue will check if the schedule context is already in the queue by examine the next pointer. Since the scheduling contexts form a circular linked list the next pointer will either point to itself, if it is the only element in that priority list, or the next element if there are more elements present. If the next pointer is null, it means that the scheduling context isn’t in the ready list and the operation will be aborted. The second step in a enqueue operation is to compare its priority with the variable \texttt{prio\_highest} and update it if the enqueued element got a higher priority. This will improve the look up performance, \(O(1)\), when the scheduler asks to retrieve the next thread to run. If the schedule context is equal to \texttt{current\_sched()}, it will be inserted into the beginning of the list in the array entry which correspond to its priority, otherwise it will be pushed into the end.

The dequeue operation will remove the schedule context from the list and traverse up the array to find the first none empty array entry with the beginner of \texttt{prio\_highest}. This is necessarily in case the highest priority scheduling context was removed.

Whenever the scheduler requests the next thread to run, the ready queue will find it in constant time by using the variable \texttt{prio\_highest} to access the correct array entry and returning the head of the list. Each thread gets a limited amount of CPU time determined by the quantum of the active scheduling context. Since the fixed priority scheduler uses Round Robin to schedule threads with the same priority, the head of the list is rotated each time the quantum of the executing thread runs out.

### 2.7.2 Scheduling Process and Lazy Queuing

When a new scheduling decision of a thread is request, the current executing thread calls the \texttt{schedule()} function located in \texttt{context.cpp} class. If the thread is calling \texttt{schedule()} within its home CPU, the basic procedure is as follows:

1. If the current thread, \texttt{current()}, is ready and has enough time quantum left, it is enqueued into the ready queue. This is necessarily because all the recent L4 kernels have the option to use lazy queuing to improve IPC performance \cite{12}.

   With lazy queuing, the ready queue can contain threads which are not ready to be executed. In the situation of thread A requesting a service from another thread B, A will send its request to B over IPC. A will be blocked and dequeued and thread B will be enqueued. B will process the request and send back the result to A. B will be dequeued whenever the IPC exchange has ended and A will have to be enqueued again. One could remove the 4 queue operations by letting the ready queue contain threads which are not ready to be executed and remove the current executing thread. It is now possible to only switch the context, to thread b, and bypass the scheduler. This makes it necessary to enqueue the current executing thread whenever a new scheduling decision is to be made.

2. A call to \texttt{next\_to\_run()} is made to retrieve the highest priority thread \( T \) in the ready queue located at the home CPU for the calling thread. Due to the lazy queuing described above, if the ready state bit of \( T \) is 0, it will be dequeued from the ready queue and \texttt{next\_to\_run()} is once again called to retrieve the highest priority thread.
3. If \( current\_sched() \neq T \rightarrow sched() \), the time quantum is replenished, if needed, and a new timeslice timeout is set. The \( current\_sched() \) is set to \( T \rightarrow sched() \).

4. Since a scheduling context may not necessarily execute its standard assigned context, due to donation, the donation chain will be traversed to find the context to execute. If no donation is required, the standard context is selected to execute.

5. The scheduling context is inserted into the ready queue and a low level (Floating-Point Unit) FPU and CPU switch is made to the active context of the \( current\_sched() \).

2.8 Timeouts

The current implementation of FIASCO supports periodic and one-shot timers. A periodic timer is generating interrupts at a fixed interval while the one-time timer only generate an interrupt for an arbitrary time point. The one-shot timer can achieve very high granularity but the hardware need to be re-programmed after each interrupt which comes with an increased overhead. The periodic timer is however set at start-up of the system and does not need to re-program itself and therefore avoids that specific overhead. However, if high granularity is needed, the period needs to be short which will increase the calls to the interrupt handler. Calling the interrupt handler in FIASCO too often may result in a significant increase in overhead if no timeouts were actually released during that interval. 

When an interrupt is generated, the interrupt handler in FIASCO will iterate through a doubly linked list, containing timeouts sorted in increasing absolute time, for expired timeouts. Whenever an expired timeout is found, the generic function expire() is called which all children of the abstract timeout class need to implement. The return value of expire() tells the interrupt handler if a reschedule is needed or not.

There are currently two types of timeouts in FIASCO, \textit{ipc.timeout} and \textit{timeslice.timeout}. The \textit{ipc.timeout} can be set for both the send and receive part of the IPC and the amount of time is set by the user. \textit{ipc.timeout} is used to avoid unnecessarily delay if any of the objects involved in the IPC become unresponsive. If a \textit{ipc.timeout} timeout expires, the IPC operation is aborted.

The second timeout is \textit{timeslice.timeout} and it is used to ensure that the scheduling context will not run out of time quantum. Each scheduling context contains a variable called left that holds the remaining quantum. Whenever a new scheduling context is marked as the current active in the ready queue, the left variable determines if the quantum should be replenished, left is equal to zero, or keep running with whatever is remaining, left larger than zero. The timeout is set as the current time + the value of the left variable of that scheduling context. If the timeout expires, the quantum will get replenished and the scheduling context will be moved to the end of the linked list, of its own priority, in the ready queue.
2.9 Inter-process Communication

In FIASCO, inter-process communication (IPC) is used to transfer data between threads, resolve hardware exceptions, faults and for virtual memory management [1]. All threads extend the sender and receive class and therefore have the ability to both receive and send messages through IPC. All IPC operations begin with a handshake to ensure that both the sender and receiver are ready to communicate. If the receiver is not ready to engage in the IPC call with the sender, the sender is added to a sender queue which is located at the receiver. The sender queue contains all threads that have requested a handshake with the specific receiver. Once a successful handshake has been performed, the sender is removed from the sender queue and the data exchange can begin. The send process is synchronous in the sense that only one sender may transmit data to the receiver at the same time and it will do so until all data are sent or an error occur.

An optimization to the IPC performance is done by letting the sender block itself whenever its waiting for a response from the receiver, bypass the scheduler and switch directly to the receiver. This shortcut will avoid the overhead of the scheduler to choose the next context to be run. It also means that priority inversion will be prevented if the receiver has lower priority than the sender because the receiver is now running with the senders scheduling context and therefore inheriting the senders priority.

There are two ways to transfer data via IPC, call and send. The difference between these is that the sender will not block after transmitting the message with the send IPC call, whereas with the call operation, the sender will be blocked until the receiver responded. An object waiting for an IPC call can do so by either performing an IPC wait or receive operation. The wait operation will accept any incoming IPC calls from any source while receive will only accept messages from a certain object. A more detailed description of the IPC in FIASCO can be found in the paper ”Verification of Fiasco’s IPC implementation” [11].

2.9.1 Thread Synchronization using IPC

Since the L4 Runtime Environment (L4Re) supports pthreads, one can use the standard mutex locks to achieve thread synchronization. It is also possible to use the regular IPC calls due the synchronous behavior. Since the icp call operation combines both the send and receive phase and the sender queue is sorted by priority.

An example to achieve synchronization is to creating a serializer thread which responds to IPC calls. The serializer will, in a loop, first perform a IPC wait followed by IPC receive. Two threads L, low priority, and H, high priority, want to access a shared resource. A and B requests a IPC call to the serializer and are placed in the sender queue. The higher priority H will first be chosen to perform the IPC call. When the IPC call from H is done, the serializer saves the capability from the IPC wait and use it in the receive operation to wait for a new message from H only. Once H is done executing the critical section an IPC send is performed to the serializer. The serializer now loops back to the IPC wait and L becomes unblocked and get access to its critical section.

Another use of IPC is the `sleep()` operation were the thread is put to sleep for a selected amount of time. It works by having the thread engage in a IPC call
to itself. The thread will be blocked and wait for a response from the sender. Because its calling itself, it will not receive any message. The only way to leave the blocked state is for the \textit{ipc\_timeout} to expire. The timeout is set to the value passed through the sleep function call.

3 Implementation

One of the challenges with implementing an EDF scheduler into FIASCO is to not break the compatibility with the existing implementation. One way to facilitate the implementation of an EDF scheduler has been suggested \cite{cite}. The idea is to divide the threads into two types, real-time and server threads. The real-time threads get assigned a period and a relative deadline and these threads will be scheduled accordingly to EDF. System threads are standard Fiasco threads in the sense that they are scheduled using the existing fixed priority scheme.

At the boot up of the FIASCO kernel, there are several threads running to initialize the system, one example of such thread is MOE which handles the initial program loading and memory management. If one would only allow real-time threads when using the microkernel for real-time purpose, the question arises of what period and relative deadline should these initial threads have? Should they terminate or keep arriving at a fixed period and do they need to be executed in a precise order in regards to each other?

The suggested solution to avoid these problems is to, instead of converting all system threads into real-time threads, allow the system to use both types of threads. This works by letting the system initially run with the standard fixed priority scheduling and then let the system designer decide, in user space, at which point the scheduler should be using EDF. Whenever the systems enter EDF mode it will prioritize real-time threads over system threads. Whenever there exists a real-time thread in the ready queue, the system will be executed accordingly to EDF. If no real-time threads exist in the ready queue, the system will start to executed the system threads accordingly to the fixed priority scheme.

To allow both kind of threads, an additional queue has been added. This queue will only contain the real-time threads and it is implemented as a min-binary heap sorted in ascending order accordingly to the threads absolute deadline.

The reason for using a binary heap over a sorted doubly linked list, where insertion runs with $O(n)$ and removal/look up is $O(1)$, is the $O(\log n)$ insertion and removal time complexity of the heap.

It is however necessarily to be able to remove not only the minimum value in the heap but also an arbitrary member which can happen if a thread becomes blocked or not ready for some reasons and therefore should be removed from the ready queue. Finding an arbitrary member in a binary heap has the time complexity $O(n)$ and removing is $O(\log n)$, this would lead to a drastic decrease in performance. However, it is possible to find an arbitrary member and remove it from the heap in $O(\log n)$ time if you save and update the position of each member in the heap array. Once having access to the position of the members in the heap, removal of a member is done by swapping the last element in the heap array with the member which is to be removed and then shift up or down depending on the value of the deleted members parent compared to the swapped
(a) Swap removed node (4) with the last node (10) in the heap array.

(b) Perform shift down because node 10 is larger than its parent node.

(c) Compare the children of node 10, if any node is smaller perform a swap.

(d) The child of node 10 is larger and therefore the deletion is completed.

Figure 3: Example of removing node 4 from the heap in $O(\log n)$ time.

With the two queues implemented, once the scheduler asks the ready queue for the next thread to schedule, it will check whether the system is in EDF mode or not. If EDF mode is enabled, the ready queue will first perform a peak into the real-time ready queue and return the thread with lowest absolute deadline. If no real-time threads exist, it will go to the system thread queue and return the highest priority thread.

It is common that a real-time thread may call system threads to perform some work for the real-time thread to advance. This leads to a system thread must have the possibility to run even if there exist real-time threads. Since FIASCO have separated the actual executing context and the scheduling context, executing context may run with different scheduling contexts. Because of that, the problem is solved by letting system threads run with a real-time threads scheduling context. Because all communication is done by IPC, which allows the receiver thread to run with the senders scheduling context, no additional changes is required.
3.1 Additional Timeouts

Two new timeout have been added to help support EDF scheduling. Both timeouts are local to the CPU core which the thread belongs to. The first timeout is called period release and all scheduling contexts have the possibility to add this timeout. It is used to keep track of thread releases and all real-time threads with a period larger than zero will activate their own period release timeout at the beginning of its release. The expire time is set to \( current.time + relative.deadline \) of the scheduling context which the thread originally owns.

Upon expire, a check is made to make sure the thread has finished its execution. If the absolute deadline is equal to zero, the thread has completed its job within its deadline since all threads that are not released have the absolute deadline set to zero. Once the preparation, see section 3.2.4 setup realtime_thread(..), for the release of the thread is done, the state of the thread will be changed to ready and put into the ready queue. An absolute deadline greater than zero indicates that the thread hasn’t finished its execution and therefore a deadline miss has occurred.

The second timeout is called deadline miss and only aperiodic and periodic threads with relative deadline less than their periods will activate this timeout upon its release. The expire time will be set to the \( current.time + relative.deadline \). It is necessarily to add this timeout since the timeslice timeout and period release are insufficient to determine a deadline miss for all cases.

An example of such case is a periodic task with relative deadline less than its period. The release timeout will only be sufficient to catch a deadline miss for tasks which are strictly periodic. The release timeout will only examine the absolute deadline at the beginning of the jobs period release, an absolute deadline with a value of zero only indicates that a job has finished its execution but do not provide any information when. Timeslice timeout will only catch a deadline miss if the job gets activated right away after being released and do not experience any interrupts or preemptions. If a job experience an interrupt or gets preempted, the remaining time quantum is saved and isn’t drained until it gets executed by the CPU again and therefore wont provide any certain information for how long the job has been in the ready queue.

3.2 Changed Made to the Schedule Context Class

This section will describe the necessarily change to the scheduling context class to allow it to both represent a system and real-time thread.

3.2.1 Additional Variables

The original scheduling context contained the variables priority, quantum and left. To be able to handle both real-time and system threads, the following variables have been added:

- **relative.deadline**: Relative deadline in milliseconds. A system thread has \( relative.deadline = 0 \) and a real-time thread has \( relative.deadline > 0 \).

- **absolute.deadline**: Deadline in absolute system time (microseconds). The thread hasn’t been released if the \( absolute.deadline = 0 \).
period: The release frequency in milliseconds. A periodic thread has period $> 0$ and an aperiodic thread period $= 0$.

in_readylist: Is the real-time thread in the ready queue?

bursts: Used by the scheduler to determine how many times the thread should repeat its job using the initial assigned absolute deadline.

heap_index: The position in the heap array and it is used to allow an arbitrary element to be removed in the heap in $O(\log n)$ time.

release_timeout: Release timeout for this scheduling context.

deadline_timeout: Deadline miss timeout for this scheduling context.

3.2.2 dominates(...) Scheduling context contains a function bool dominates(sched_context *sc) that compares the priority of the calling scheduling context with another scheduling context sc. The return value is true if the priority is strictly higher than sc, otherwise false. Dominates is currently only used by the ready queue deblock operation to determine if the scheduler need to perform a new scheduling decision after a scheduling context has been inserted into the ready queue. Since real-time threads do not use the priority variable, the function has been modified to allow support for comparison between system and real-time threads. The relative deadline is used to distinguish between a real-time and a system thread. The relative deadline is always zero if the scheduling context is a system thread, otherwise it will be treated as a real-time thread. If both the calling scheduling context sched and sc are of type real-time, the smallest absolute deadline will determine the result. It is sufficient to only compare the absolute deadline and ignore the relative deadline because the comparison will always be made between released scheduling contexts. All released real-time scheduling contexts have an absolute deadline assigned to them. If a real-time and system thread are compared, the real-time thread will always dominate the system thread.

3.2.3 in_ready_list() To determine if a scheduling context is in the ready queue the function bool in_ready_list() is used. It returns true if the scheduling context is in the ready list and false if not. Due to the second ready list for real-time threads, the type of the scheduling context is determined by looking at the relative deadline. In the case of a system thread, the next pointer, described in Sec. 2.7.1, will tell if the scheduling context is in the ready queue or not. In the case of a real-time thread, the variable in_readylist indicates if the scheduling context is in the ready list or not.

3.2.4 setup_realtime_thread(...) The release of a real-time thread requires a setup process which assigns an absolute deadline to the scheduling context, sets up the period release and/or deadline miss timeout. The type of timers needed depends on if the thread is strictly
periodic or aperiodic. To handle the setup, a function `void setup_realtime_thread()` has been added to the schedule context and it is called at the beginning of each release before the schedule context is inserted into the ready queue.

The setup process of a real-time thread when the system is in real-time mode is the following:

1. If the period is larger than zero, the schedule context is periodic and therefore requires a release timer to maintain a periodic release pattern. The timeout is set to `current_time + period`.

2. If the relative deadline is less than the period, i.e. the thread is not strictly periodic, or the period is equal to zero, i.e. the thread is aperiodic, a deadline timeout is required to keep track of a possible deadline miss. The timeout is set to `current_time + relative_deadline`.

3. The absolute deadline of the scheduling context is set to `current_time + relative_deadline`. The time quantum is replenished by the value of its relative deadline.

If the setup is made in system mode, the absolute deadline is set to zero and any existing release and deadline timeouts are removed. The reason behind this is described in Sec. 3.3.1.

### 3.2.5 change_thread_param(...) 
To allow a thread to change from system to real-time thread or vice versa, a function `void change_thread_param(Unsigned32 option1, Unsigned32 option2, bool realtime)` has been implemented. If `realtime` is true, the `option1` and `option2` represent the new period and the relative deadline, otherwise it represents the new priority and quantum. Whenever a change is made to the parameters of a real-time scheduling context, a restart to all timing constraints for the thread is made. Changing parameters or converting a system to real-time thread therefore requires extra precaution by the end system designer. To change any parameters for a schedule context, the following steps are performed:

1. The release and deadline timeout are removed from the timeout queue and the scheduling context is dequeued.

2. If `realtime = true` the period is set as `option1`, relative deadline as `option2`, quantum as `option2` and the function `setup_realtime_thread()` is called.

3. If `realtime` is false, the period, absolute and relative deadline is set to zero. The priority is set as `option1`, quantum as `option2` and the time quantum is replenished.

4. If `current_sched() = this`, a new the timeslice timeout is programmed for `current_time + quantum` to avoid a false timeslice_timeout.

5. The scheduling context is enqueued into the ready list.

After a call being made to `change_thread_param(...)`, a request to the scheduler is made to make a new scheduling decision in case of priority change in the ready queue.
3.2.6 set(...) 
The `int set(L4_sched_param const *p)` is used to initialize the scheduling context. This function is called whenever a new thread is about to run for the first time. As argument it takes a structure called `L4_sched_param` containing the wanted properties for the specific type scheduling context. The set function has been modified to also be able to handle scheduling parameters for real-time threads.

To determine whether its a real-time or system thread, the `sched_class` attribute is examined. If `sched_class` $\geq 0$, the scheduling context is a system thread and the priority and quantum is assigned. If `sched_class` $= -2$, the scheduling context will be assigned the real-time thread parameters and the `setup_realtime_thread(...)` function is called.

3.2.7 Destructor 
Due to the introduction of two new timeouts, a destructor has been added to the scheduling context class. Once called, the destructor will make sure that the release and deadline miss timeout are removed from the timeout list. A last check to make sure the scheduling context is removed from the ready queue is also done.

3.3 Changes to the Ready Queue 
This section describes the changes made to the ready queue class in order to support EDF scheduling. A similar queue is added to the IPC sender queue in order to support synchronization with both system and real-time threads.

3.3.1 System and Real-time Mode 
The ready queue can now operate in two different modes, system and real-time. In system mode, the function `next_to_run()` will always return the highest priority scheduling context in the fixed priority ready queue. If real-time threads are created during system mode, they will be added without any release and deadline timeout to the EDF ready queue. The absolute deadline will be set to zero to indicate that it was added during system mode and it will not be chosen by the scheduler. Once the ready queue is changed to real-time, all real-time scheduling context inside the ready queue with `absolute_deadline = 0` will get the relevant timeouts activated and a new absolute deadline assigned by call the `setup_realtime_thread()` function. Its important to note that the absolute deadline, for the first release of a thread that were added during system mode, is set from the time of change to real-time mode and not when thread was added to the ready queue during system mode.

3.3.2 Additional Variables 
As described in the Ready Queue section, the fixed priority ready queue structure is made up by `prio_highest` and a circular linked list array. With the binary heap implementation, the following variables have been added:

- **length**: Number of elements in the heap array.
• **array_size**: Current allocated memory in the array.

• **heap_array**: Array of scheduling contexts representing the heap.

• **realtime_scheduling**: Is ready queue in real-time mode?

### 3.3.3 set_scheduling_realtime_mode(...) 

Allows the end system designer to change the ready queue to be in either system or real-time mode. The function is of type `void` and takes one Boolean value named `enable` as argument. If a change is made from system to real-time mode, the procedure is:

1. Set `realtime_scheduling = true` to make `next_to_run()` return real-time threads if present in the ready queue.

2. To handle all real-time threads that have been added during system mode, `next_to_run()` will be called until `relative_deadline > 0` or `absolute_deadline > 0`. The returned threads by `next_to_run()` is dequeued and calls `setup_realtime_thread()`. The threads are then enqueued again.

If a change is made from real-time to system mode, `realtime_scheduling` is set to false and it is up to the end designer to make sure all real-time threads have finished their work and their period is set to zero before a change is made to avoid false deadline misses and more releases.

### 3.3.4 Enqueue and Dequeue

The enqueue and dequeue operations are used to insert and remove scheduling context from the ready queue. Since all scheduling contexts contain parameters for both system and real-time scheduling it is not necessarily to have two separated enqueue and dequeue functions for both types. One can determine the type of thread by looking at the relative deadline since it will always be strictly larger than zero if it is a real-time thread.

For the heap queue, two helper functions `shift_up` and `shift_down` have been added. The `shift_up` starts at a given location and switch the selected child with its parent as long as the child has an earlier deadline than its parent. `shift_down` begins with a given node and as long as the selected node has a larger absolute deadline than the child with the earliest deadline, the selected node and that child will be swapped. In both functions, the index of the scheduling context, in the heap array, is saved in the `heap_index` variable to make it possible to remove an arbitrary scheduling context in $O(\log n)$ time.

If a enqueue operation is requested on a real-time thread, the insertion will be made at the end of the heap array followed by a shift up operation. If the request is made on a system thread, the scheduling context, `sched`, will be inserted at the front of the linked list corresponding to its priority if `sched = current_sched()` otherwise at the back. The `prio_highest` is updated if `sched` has higher priority.

A dequeue of a system thread is done by removing the scheduling context from the linked list and updating the `prio_highest`. The update is done by, starting with the old value of `prio_highest`, iterating down the linked lists until a non-empty list is found. Removing a real-time thread with the earliest absolute deadline is done by swapping it with the last element in the heap array and
performing a shift down operation. A removal of an arbitrary element, \( e \), from the heap is done by replacing \( e \) with the last element in the heap array. If the new element \( e \) has an earlier deadline than its parent, a shift up operation is performed from the position of \( e \), otherwise a shift down.

3.4 User Calls to the Kernel

To still maintain the idea of having the basic implementation, in the kernel space, the user calls made to the kernel scheduler have been designed such that addition scheduling algorithms can be implemented in user space.

3.4.1 l4.scheduler_run_thread_edf(…)

Start executing a real-time thread.

**Arguments:**

- \( \text{scheduler} \): Capability to the scheduler object.
- \( \text{thread} \): Capability of the thread to be executed.
- \( \text{sp} \): Scheduling parameters.

The \texttt{l4.sched.param.edf.t} is a new data structure containing the CPU affinity, period and relative deadline. When \texttt{l4.scheduler_run_thread_edf(…)} is called, an IPC message containing the scheduling data will be sent to the scheduler. The new thread will be prepared and added to the ready queue. If a request is made to run a real-time thread during system mode, it will not be setup or executed until a change to real-time mode is done.

3.4.2 l4.scheduler_work_done(…)

The end system designer can alert the scheduler that a periodic or aperiodic real-time thread is done with its work and should go to sleep until its next release.

**Arguments:**

- \( \text{scheduler} \): Capability of the scheduler object.
- \( \text{thread} \): Capability of the thread to be executed.

If called inside an aperiodic task with no bursts, the thread will be blocked until a manual release is made from user-space, see Sec. 3.4.4. If the sporadic task came with bursts, the call isn’t blocking until the function has been called \( n \) times where \( n = \text{burst size} \). If the thread is periodic, it will be blocked until the next \texttt{release.timeout} associated with the thread has expired.

3.4.3 l4.scheduler_set_thread_param(…)

Change the parameters of the scheduling context, e.g. system to real-time thread.

**Arguments:**

- \( \text{scheduler} \): Capability of the scheduler object.
• **thread**: Capability of the target thread.

• **option1**: Period if `realtime = true` else priority.

• **option2**: Relative deadline if `realtime = true` else quantum.

• **realtime** Does options represent a real-time thread?

Once called, all timeouts will get a reset and the time quantum is replenished. A new schedule decision is made.

### 3.4.4 `l4_scheduler_release_thread(...)`

Request a single or burst release of an aperiodic thread.

**Arguments:**

- **scheduler**: Capability of the scheduler object.

- **thread**: Capability of the thread to be executed.

- **bursts**: The burst size of the release.

Can only be used for aperiodic threads. Once called, a new schedule decision is made.

### 3.4.5 `l4_scheduler_set_realtime_mode(...)`

Change the scheduling mode of the ready queue.

**Arguments:**

- **l4_cap_idx_t scheduler**: Capability of the scheduler object.

- **bool enable**: True for real-time mode. False for system mode.

If changed from system to real-time mode, all real-time threads added during system mode will be prepared for execution and released. A change from real-time to system mode requires all real-time thread to be terminated before called.

### 3.5 Create a Sporadic Task with Accumulated Bursts

Since a microkernel should only contain the basic functionality, the EDF implementation does not support sporadic tasks. However, it does support aperiodic behaviour and with the available IPC calls to the kernel it is possible to create sporadic tasks in user space. This following section will give an example of how to achieve a sporadic task with the ability to be released in bursts.

There are different ways of handling a release of a sporadic task before its minimum inter-arrival time. In this example, a sporadic task will accumulate releases as bursts if a release request is done before the minimum inter-arrival time. An external source will generate an interrupt that leads to a function call, `release_thread(...)`, to request a release of a sporadic thread.

**Pseudo Code: Creation of Sporadic Task**
To create a sporadic task in user space, the thread will be of type real-time thread with a period of zero, to make it aperiodic. Having an aperiodic thread will result in no release timeouts and releases can therefore be decided in user space.

To setup the aperiodic thread, it is started by the interrupt handler with `l4_scheduler_run_thread_edf(...., AperiodicThread, 0, relative_deadline)`. Since no actual release of the sporadic task is made during the setup process, once the initial preparation of the thread is done, e.g. variable creation, a call to `l4_scheduler_work_done(....)` should be made to put it to sleep until the next release.

Once a sporadic release request is made by the interrupt handler, last release time + minimum inter-arrival time is compared to the current time to prevent a violation against the minimum inter-arrival time. If no time violation is made, a call to `l4_scheduler_release_thread(...., thread, Bursts)` is made. The aperiodic thread will ignore the `l4_scheduler_work_done(....)` call `bursts` number of times and therefore perform the work multiple times with the same absolute deadline. If a release is made before the minimum inter-arrival time, the burst size will be incremented and no release call is made.
4 Testing the Implementation

To verify that the EDF implementation is functioning correctly, several tests have been designed and executed. Most of the tests are made and executed in user space, however a few tests require small modifications to the kernel.

4.1 Timing

To make sure that the setup of timers is working as intended, both deadline and release timeout is setup the same way, a test has been designed to measure the time between each release. The L4 environment of FIASC0 has a function \texttt{l4_rdtsc()} to read the internal CPU time stamp counter. The CPU time stamp is converted to nanoseconds with the help of \texttt{l4_tsc_to_ns(\ldots)}.

The test is performed by creating two periodic threads with a period and deadline of one and two seconds. At each new release, the CPU time stamp is saved and the previous time stamp is subtracted to the current time stamp. The stamp value is converted to nanoseconds and printed out to the console. To keep the CPU busy, each thread performs a small work load.

The output will experience various degrees of oscillations depending on if the hardware timer is in periodic mode or one-shot mode and the resolution of the timer used to generate the interrupt. These oscillations can be seen in the example output below. The output is made by two periodical threads with period 1 and 2 seconds with the APIC timer running in periodical mode.

Example output:

<table>
<thead>
<tr>
<th></th>
<th>Thread 1 Time: 1005822368ns</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Thread 1 Time: 1000198336ns</td>
</tr>
<tr>
<td></td>
<td>Thread 2 Time: 2004234496ns</td>
</tr>
</tbody>
</table>

4.2 EDF Queue

Both the ready queue and the sender queue use the same min-binary heap structure. To test the heap implementation, \( n \) random periods are generated from an interval of 300-1000ms. Pthreads are used in this test and the period is passed along as argument.

The test starts by creating \( n \) system threads. Each thread changes its scheduling parameter to real-time and assigns its absolute deadline and period to the random generated period. Since the scheduler is still in system mode, each thread will be blocked after it has changed to a real-time thread. When all threads are created and converted to real-time threads, the system changes to real-time mode. Once the change to real-time mode is made, the absolute deadline for each real-time thread is set. When threads start their execution, they save their absolute deadline and execution order to an array. At the end, the array is checked so the threads executed in ascending order with respect to their absolute deadline.

4.3 Schedule Calls

Once a real-time thread has been released, no more scheduler decisions should be made until the thread work is done and if no new releases occur. The fixed point scheduler uses the timeslice timeout to periodically interrupt the current
executing thread to achieve time sharing. These interrupts should not be present in the case of executing a real-time thread.

The test compose of:

1. A print statement is added to the beginning of the function `schedule()` in `context.cpp`. The print statement adds the output "Scheduled Called!" to the console whenever a new schedule decision is being performed.

2. A periodic real-time thread is created by the main thread with a period and relative deadline of 5 seconds. The thread contains a print statement to inform that it has started its execution of work followed by a simulated work load to keep the CPU busy. The work load is inside an infinite loop and the thread will therefore be executed until a release with an earlier deadline or a deadline miss has occurred.

3. The main thread calls `l4_scheduler_run_thread_edf(...)` to start executing the periodic thread. However, since the system is still in system mode, the thread will get its parameters assigned and enqueued into the ready queue without executing its work.

4. A call to `l4_scheduler_set_realtime_mode(...)` is made and the periodic thread will start to execute its work. Since changing the scheduling mode will result in a new scheduling decision, the "Scheduled Called!" should be seen in the console.

5. The periodic thread will print to the console that it has started and no more calls to the scheduler should be made. After 5 seconds the thread will exceed its deadline and an assertion will be made to alert the user about the deadline miss.

**Pseudo Code:**

```c
void thread_1()
{
    print(Thread 1 started ...)
    while(1)
    {
        // Simulated work to keep the CPU busy.
        simulated_work();
    }
}

void main(){
    // Set the parameters to period and deadline 5sec
    l4_sched_param_edf_t sp = l4_sched_param_edf(5000, 5000);
    // Execute the thread
    l4_scheduler_run_thread_edf(Thread_1, sp);
    // Change to real-time mode
    l4_scheduler_set_realtime_mode(..., 1);
    sleep();
}
```

The expected outputs to console:

```
Scheduled Called!
Thread 1 started − No more schedule calls should be made
####### Deadline Miss, quantum 0 ####### relative_deadline: 5000
Assertion failed at .../src/kern/timeslice_timeout.cpp:46: false
```
4.4 IPC Sender Queue

To test the implementation of the min-heap for real-time threads added to the sender queue, three real-time threads were created with different relative deadline as well as a system thread with the ability to receive IPC calls. The high priority thread performs an IPC call to the system thread and keeps it busy while the remaining threads are placed into the sender queue. With the help of sleep() in the medium thread, the low priority thread performs the IPC call before the medium priority to make sure that the order is in EDF. The success of the test is determined by the order of which the IPC calls are made. To keep track of the order, the threads sends a text message to the console when the IPC call is done. A successful test is done when the highest priority thread finish first followed by the medium and then the low priority.

Pseudo Code:

```c
void thread_system(){
    // Wait for the initial IPC call by Thread_High
    l4_ipc_wait (...);
    // Sleep for 3 seconds to allow the other IPC calls.
    sleep(3);
    while(1)
    {
        // Reply and send back a response to the sender.
        l4_ipc_reply_and_wait (...);
    }
}

void thread_high(){
    // Period 100 seconds, Relative Deadline 3.7 seconds
    l4_ipc_call (thread_system , ...);
    print("High Priority Thread Done");
    l4_scheduler_work_done (...);
}

void thread_medium(){
    // Period 100 seconds, Relative Deadline 4 seconds
    sleep(1);
    l4_ipc_call (thread_system , ...);
    print("Medium Priority Thread Done");
    l4_scheduler_work_done (...);
}

void thread_low(){
    // Period 100 seconds, Relative Deadline 5 seconds
    l4_ipc_call (thread_system , ...);
    print("Low Priority Thread Done");
    l4_scheduler_work_done (...);
}

void main(){
    l4_scheduler_run_thread (thread_system);
    l4_scheduler_run_thread_edf (thread_low);
    l4_scheduler_run_thread_edf (thread_medium);
    l4_scheduler_run_thread_edf (thread_high);
    // Change to real−time mode
    l4_scheduler_set_realtime_mode (... , 1);
}
```
4.5 Aperiodic Releases

The custom releases of aperiodic threads are tested by creating two aperiodic threads and one periodic. One of the aperiodic threads will be released with a burst size of 1 and the other one with no bursts. Both of the aperiodic threads will perform around 45ms of work and update a global state when they finish the work. The periodic thread will act as the thread releaser and all threads will be converted to system threads and be put to sleep once done to also test the change of thread parameters.

The test composes of:

1. Main thread sets up and makes the threads ready to execute. To allow the aperiodic threads to be setup for execution, reach the first work done call, the periodic thread has a period and deadline of 5 seconds. The aperiodic thread with bursts and no bursts has a relative deadline of 150 and 70ms.

2. The main thread changes the system to real-time mode, both the aperiodic threads will start to execute and be put to sleep until the next release call. The periodic thread will change its relative deadline to 20ms to allow a release of both the aperiodic threads without an interruption. It is then converted to a system thread and put to sleep.

3. The aperiodic thread with no bursts will execute first due its earlier absolute deadline. Once the 45ms work is done, it increments its own global work done variable. The global work variables are examined to make sure that it was released before the aperiodic thread with bursts.

Pseudo Code: Aperiodic Release Test

```cpp
// Period 5000ms, Relative Deadline 5000ms
// Change the relative deadline to 20ms and period to 5 sec.
l4_scheduler_set_thread_param(5000, 20, 1);
// Release the aperiodic thread with burst size 1.
l4_scheduler_release_thread(thread aperiodic_1, 1);
// Release the aperiodic thread with no bursts.
l4_scheduler_release_thread(thread aperiodic_0, 0);
// Put to sleep until next periodic release.
l4_scheduler_work_done(thread periodic);
// Change to system thread
l4_scheduler_set_thread_param(5, 50, 0);
l4_sleep_forever();
}
// Period 0 , Relative Deadline 150ms
void thread aperiodic_1{
int bursts = 0;
l4_scheduler_work_done(thread aperiodic_1);
while(1){
// keep the cpu busy for 45ms of work.
simualte_work(45);
if(burst > 0)
break;
Work_Done_1++;
l4_scheduler_work_done(thread aperiodic_1);
}
```
Work Done 1++;  
// Check if the test was successful.
if (Work Done 0 != 1 || Work Done 1 != 2)
   Success = 0;
else
   print( "Success!" );

l4_scheduler_work_done(thread_a periodic_1);
l4_scheduler_set_thread_param(thread_a periodic_1, 5, 50, 0);

// Period 0, Relative Deadline 70ms
void thread_a periodic_0{
l4_scheduler_work_done(thread_a periodic_1);
while(1){
   // keep the cpu busy for 45ms of work.
simualte_work(45);
   // Increment the number of work done.
   Work Done 0++;

   l4_scheduler_work_done(thread_a periodic_1);
}

Work Done 0++;
if (Work Done 1 != 0 || Work Done 0 != 1)
   Success = 0;

l4_scheduler_work_done(thread_a periodic_0);
l4_scheduler_set_thread_param(thread_a periodic_0, 5, 50, 0);
}

void main(){
l4_scheduler_run_thread_edf(thread_periodic);
l4_scheduler_run_thread_edf(thread_a periodic_1);
l4_scheduler_run_thread_edf(thread_a periodic_0);
l4_scheduler_set_realtime_mode(1);
l4_sleep_forever();
}

4.6 Basic Pthreads

To test pthreads, 20 threads are created using pthreads. The task of each thread is to perform a 50ms workload and increment a counter inside a critical section 10 times. The critical section is created by a pthread mutex lock. All pthreads created are by default using fixed priority scheduling. To create a real-time pthread, each pthread begins by calling l4_scheduler_set_thread_param(...) which allows the thread to change scheduling parameters. Once the thread makes the call, it will be put to sleep because the ready queue is still in system mode and the main thread can continue releasing threads. When all pthreads are created a call is made to the scheduler to request a change from system mode to real-time mode and the threads will start to execute. The test is considered successful if the final value of the counter is 200 since the mutex lock creates a critical section to prevents interrupts while incrementing the shared resource Counter. Executing the test several times shows that mutex locks are supported when using real-time threads.
Pseudo Code: Basic Pthread Test

```c
// Convert to real-time thread with period and deadline of 1.2 sec
l4_scheduler_set_thread_param(..., 1200, 1200, 1);

for (int n = 0; n < 10; n++){
    // Execute work for 50ms
    work_load(50);
    // Begin Critical Section
    pthread_mutex_lock(&Mutex);
    Counter++;
    pthread_mutex_unlock(&Mutex);
    // End Critical Section
    l4_scheduler_work_done(..., pthread_l4_cap(pthread_self()));
}

void main(){
    for(int i = 0; i < 20; i++)
        pthread_create(&Threads[i], NULL, Worker, NULL));
    // Allow all the real-time pthreads to start execute
    l4_scheduler_set_realtime_mode(..., 1);
    for(int i = 0; i < 20; i++)
        pthread_join(Threads[i], ...)
    if (Counter == 200)
        printf("Success\n");
    else
        printf("Failed\n");
}
```

5 Benchmark: EDF vs RM

The performance of the EDF implementation is compared to the already existing fixed priority scheduling in FIASCO. To facilitate the creation of the benchmark test, the fixed priorities are set accordingly to RM scheduling. It is also interesting to compare EDF and RM since it has been discussed before if the common conception that EDF got higher overhead than RM is true or not [3].

The benchmark test consist of letting different amount of threads $S_{threads} = \{10, 40, ..., 160, 190\}$ execute a workload which corresponds to a certain CPU utilization. The utilization $U$ for a thread $i$ set is set to $U = \frac{target \ utilization}{number \ of \ threads}$. The computation time $C_i$ for a thread is set to $C_i = U \times T_i$. Each thread is strictly periodical and will be released 5 times.

For example, if a benchmark is performed with 50% CPU utilization, the benchmark starts by using 10 threads. To decide how long each thread should be executed on the CPU, the target utilization is set to 0.50, since the CPU utilization for this benchmark is set to 50%, and $U = \frac{0.50}{10} = 0.05$. If a thread $i$ has a period of 1000ms, its execution time $C_i$ will be $U \times T_i = 0.05 \times 1000ms = 50ms$. Accordingly to the thread set $S_{threads}$, the benchmark is first perform by 10 threads. Once the benchmark for 10 threads is completed, the benchmark is executed again but with 40 threads. The execution is repeated but with an increment, of threads used in the benchmark, by 30 until 190 threads has been used.
The test is performed on an emulated uniprocessor (ia32) system using QEMU. The processor is running at 2494MHz and 255MB RAM is available at boot and the APIC timer is running in periodic mode. The performance was measure by letting the relevant kernel functions, such as `schedule()` and enqueue/dequeue, measure its execution time and calls received. The time was measured with the CPU time stamp counter (TSC) register existing on the x86 processors that should offer low overhead and high resolution.

Two new IPC calls to the scheduler have been implemented to facilitate the benchmark tests:

- `l4_scheduler_record_benchmark(capability scheduler, bool enable)`
  Enable or disable recordings of all the benchmark data.

- `l4_scheduler_print_benchmark(capability scheduler)`
  Prints out the result from the recorded benchmarks. After printing out the benchmark data, all benchmark data is set back to zero.

The workload created to match a certain utilization percentage is generated by repeatedly performing basic math operations on a volatile variable. By measuring, using TSC, how long it takes to execute a number of iterations, an estimate can be derived of how many iterations needed for the workload to correspond to a certain amount of CPU time. The resolution of the measurements is in microseconds. It is important to have in mind that the result isn’t perfectly deterministic but the resolution is good enough for the purpose of this benchmark.

The benchmark was designed to test out the average case by randomly assign periods in the range of 300 to 1000ms. The priority for the threads running under RM is set by sorting the threads with respect to their relative deadline and assigning decreasing priorities, starting from 254, to each thread in the sorted order. To compensate for timeslice timeouts in RM scheduling, the number of calls to requeue have been subtracted from the number of schedule calls. This was done because the default timeslice for fixed priority is set to 10ms and each timeout requires a schedule call as well as replenish the time quantum. If there is only one thread in the active priority list, it will get interrupted every 10ms and a new schedule call is made. This could be misleading for the end result and therefore is removed from the total schedule calls.

A special idle thread is created for the benchmark tests to avoid repeatedly calling the schedule and enqueue/dequeue whenever there are no threads ready to be executed. The standard time quantum is 10ms and therefore calls to the schedule will be made every 10ms when no thread is ready to be executed except the idle thread. To avoid this interference, the new idle thread has a very large time quantum such that no timeslice timeout will be fired whenever the idle thread is running.

The benchmark is created using pthreads and the recording of data starts when all the threads in one set have been created and scheduling parameters been set. The recording stops when all threads in that set have been joined. This procedure is repeated for each amount of threads in $S_{threads}$ for the utilization $\{10, 40, ..., 160, 190\}$. The difference between the results for RM and EDF is calculated as $(EDF - RM)/RM$. 

28
5.1 Benchmark Result

This section contains the result of the benchmark between EDF and RM. The figures shows the mean value from five executed tests.

Figure 4: Improvements in schedule calls.
Figure 5: Improvements in enqueue/dequeue Calls.

Figure 6: Improvements in enqueue time for RM.
Figure 7: Improvements in enqueue time for EDF.

Figure 8: Improvements in dequeue time for RM.
6 Alternative Implementation of EDF

There have been many discussions about the actual overhead of EDF compared to other scheduling policies [3]. The reason for the extra overhead in EDF compared to, for example, fixed priority are the $O(\log n)$ time complexity for insertion and removal from ready queue and the need to re-calculate the absolute deadline at the beginning of each job release. The fixed priority scheme with a fixed number of priorities has constant insertion and removal time complexity in regards to the ready queue and no extra overhead at each job release. One idea of decreasing the overhead of EDF is to use fixed priority to mimic the behaviour of EDF scheduling. The ideal solution would be to run EDF scheduling with the same time complexity as fixed priority.

6.1 Fixed Priority with Priority Promotions

A method to achieve EDF behaviour for fixed priority in both uniprocessor and multiprocessors, called Fixed Priority with Priority Promotions(FPP), has been proposed in [9]. The idea of FPP is to increase the priority of each job at fixed time points to make the priority of a task to be set accordingly to EDF. The policy for when a job should get promoted and with what priority is called Increase Priority at Deadline Difference(IPDD). To find out the promotion points and priorities, the set of tasks, $t_1, t_2, ..., t_{i-1}$, is sorted in ascending order with regards to their relative deadlines. At each job release, $j_i$, of task $i$, $j_i$ will get assigned the initial priority level $i$ and will be promoted $i - 1$ times. The highest priority level is 1 and each task have the worst execution time $C_i$, relative deadline $D_i$ and minimum inter-arrival time $T_i$ parameters.

![Figure 9: Improvements in dequeue time for EDF.](image)
assigned to them. The priority promotions for a job \( j_i \) are performed at time 
\([r_i + D_i - D_{i-1}], (r_i + D_i - D_{i-2}), \ldots, (r_i + D_i - D_1)\) where \( r_i \) is the release time of job \( i \).

Since the most expensive part of FPP is the priority promotions, an offline test has been designed to determine if it is possible to schedule a set of tasks with standard FP. If the set of tasks isn’t schedulable, the test will start to assign priority promotions to tasks until the task set become schedulable and all tasks meet their deadline. The number of priority promotions can therefore be decreased since not all tasks need to be promoted and overhead is avoided.

It has been shown that FPP running with the IPDD policy generates EDF scheduling and simulations confirms the lower overhead compared to standard EDF [3]. However, in a real kernel, the promotions points have to be generated at the specific time intervals with some kind of interrupts.

One way of generating the promotions in FIASCO is to implement promotion points as timeouts. The heavy use of timeouts may generate significant overhead because insertion to the timeout queue isn’t done in a constant time and depending if the kernel uses a one-shot or a periodic timer to check the timeout queue, additional overhead will most likely be present. If a one-shot timer is used a new setup of the one-shot timer is required after each expired timeout. The additional overhead required to set up the timer is dependent on the hardware and therefore it has to be chosen carefully. The use of a periodic timer requires no overhead after each interrupt. Though getting a fine enough granularity for the promotion points will most likely lead to additional overhead due to checking empty timeout queues.

6.2 Mimic EDF with Fixed Priority using Dynamic Counters and Worst Case Ready Queue

An approach to create a method to mimic EDF scheduling using fixed priority was looked into using dynamic counters [6]. Dynamic counters are used to bound future event releases accordingly to a set of upper and lower event arrival curves. It was originally developed to save energy by dynamically adjusting the power supply in embedded systems such that the real-time constraints are met with a minimum energy consumption.

The work of dynamic counters has been used in a scheme [7] to control the heat generated by the processor, using Dynamic Voltage Frequency Scaling, in a time constraint system. With the help of dynamic counters, by creating an upper bound for present and the future workload of jobs, a worst case ready queue (WCRQ) for job releases is created. Each job in the WCRQ gets assigned a frequency which the processor core will execute that specific job with such that the lowest possible frequency is used in regards to the system energy efficiency, temperature and timing constraints. The idea to mimic EDF with fixed priorities is to use the WCRQ and instead of assigning each job a frequency, each job gets a priority such that all jobs will be executed accordingly to EDF.

6.2.1 Dynamic Counters and WCRQ

The dynamic counter \( DNC_{i,j} \) gives an upper bound of job releases for task \( i \) at present time. The value of \( DNC_{i,j} \) depends on a set of staircase curves defined
as: \( \alpha_{i,j}(\Delta) = (N_{i,j} + \lfloor \frac{\Delta}{\delta_{i,j}} \rfloor) \). The \( N_{i,j} \) variable is the maximum burst capacity, \( \delta_{i,j} \) is the step rate of staircase curve \( j \), i.e. release frequency of jobs from task \( i \), and \( \Delta \) is the delta time.

The update of the dynamic counter can be seen in Algorithm 1.

**Algorithm 1 Update of DNC**

1. **procedure** UpdateDNC
2. if \( s == \text{event arrival} \) then
   3. if \( DNC_{i,j} == N_{i,j} \) then
      4. SetTimeout(\( \text{CurrentTime} + \delta_{i,j} \))
   5. \( T_{i,j} = 0 \)
   6. \( K_{i,j} = 0 \)
   7. \( DNC_{i,j} = DNC_{i,j} - 1 \)
   8. if \( s == \text{expired timeout} \) then
      9. \( DNC_{i,j} = \min(DNC_{i,j} + 1, N_{i,j}) \)
   10. \( K_{i,j} += 1 \)
   11. SetTimeout(\( \text{CurrentTime} + \delta_{i,j} \) )

On line 3, a renewal point has occurred and it is a condition where the past becomes irrelevant for future releases. It is due to the maximum burst capacity \( N_{i,j} \) has been reached and therefore the "memory" can be set to 0. Variable \( T \) on line 5 gives the time since last renewal point.

To get the number of future releases of task \( i \) for the time interval \( t \) to \( t + \Delta \), the following equation is used:

\[
\text{frel}_{i,j}(\Delta) = \begin{cases} 
\frac{\Delta+T_{i,j}-K_{i,j}\delta_{i,j}}{\delta_{i,j}} & \text{if } DNC_{i,j} < N_{i,j} \\
\frac{\Delta}{\delta_{i,j}} & \text{if } DNC_{i,j} = N_{i,j}
\end{cases}
\]

where \( K_{i,j} \) is the smallest integer value such that \( K_{i,j} \cdot \delta_{i,j} \leq T_{i,j} \).

By adding all the jobs that may be released, in regards to \( a_{i,j} \) at the present time, \( DNC_{i,j} \), with the future number of possible releases, equation 1, the upper bound releases for period \( t \) to \( t + \Delta \) can be defined as:

\[
F_{i,j}(\Delta) = DNC_{i,j} + \text{frel}_{i,j}(\Delta)
\]

To find the upper bound of job releases of task \( i \) in the interval \( t \) to \( t + \Delta \), with regards to a deadline \( D_i \), a maximum bound for pending jobs \( B_i \) and taking all the staircase curves \( j \) into account, the equation can be defined as:

\[
F_{i}(\Delta) = \min_{j \in J}(\max\{F_{i,j}(\Delta - D_i), F_{i,j}(\Delta) - B_i\})
\]

For an interval \( t \) to \( t + \Delta \), job that has not yet been released but have the possibility of getting released within that interval are called potential jobs. A potential job that got released is called a real job. The WCRQ can contain both potential and real jobs and is populated by Algorithm 2.
Algorithm 2 Populate the WCRQ with potential jobs

1: procedure PopulateWCRQ(WCRQ, horizon)  
2:   for $i = 1$ to $N$ do  
3:     $x = 0$, $sum = 0$  
4:       do  
5:         $new = F_i((x+1) \times D_i) - sum$ \Comment{Equation 3}  
6:         $d = now + (x+1) \times D_i$  
7:         for $j = |WCRQ|$ to $new + |WCRQ|$ do  
8:           $WCRQ = (i, C_i, d, potential, sum++)$  
9:         $x++$  
10:     while $x \times D_i \leq horizon$

The horizon decides how far into the future, from current time point, potential releases of jobs should be added to the WCRQ. Since $F_i(\Delta)$ gets the number of potential releases at the interval $[Now, Now + \Delta]$ and $\Delta$ will take the value of the first worst case deadline and then the second duplicates of potential jobs will be created. The $sum$ variable helps to remove the duplicates by subtracting the previously added jobs.

6.2.2 An Approach to Mimic EDF using Fixed Priority

One approach to run EDF with fixed priority is to use algorithm 2 to generate priorities for each job release. The population of the WCRQ is an expensive operation because it is necessary to loop over all tasks and add the worst case deadline until $currenttime + horizon$, therefore it is not feasible to do this at each job release. One possible way to decrease the amount of tasks to create during the population of the WCRQ is to create two queues. One queue is the WCRQ for sporadic tasks and is populated by algorithm 2. The second queue only contains periodical tasks and the queue is re-created whenever a new task is added to the task set.

The periodical WCRQ can be populated by simulating job releases and execution for all periodic tasks over the hyperperiod for that periodic task set. The simulation is done using EDF and the priority for each job is assigned in regards to the result of the simulation such that it mimics the simulated behaviour. The job priorities for each release have to be stored, for example in an array where each position represents the release number for that task, such that a priority look up can be performed at each job release to assign the fixed priority.

The sporadic WCRQ is generated by algorithm 2. It is not feasible to generate the WCRQ at each job release. Because the queue represents the worst case scenario it is very likely that the queue will still be valid after several job releases. The queue is called valid as long as the priority assigned by the WCRQ still mimic EDF behaviour. The queue will be valid as long as the difference between the absolute deadline of a potential job and the actual absolute deadline of that same job is smaller than the smallest absolute deadline difference between any two potential jobs.
Figure 10: An illustration of the smallest absolute deadline difference.

Figure 11 shows an example where a release of a task, T2 (red box), is still valid after experiencing a delayed release due to its sporadic behaviour. One of the main questions is how large the horizon should be. Having it too narrow will increase the frequency of creating a WCRQ and therefore increase the overhead. Having it too wide will increase the chance of the WCRQ to get invalid and increase the possibility of a priority overflow.

Whenever a new schedule decision is to be made, the highest priority job from each WCRQ is fetched and their absolute deadline are compared. The job with the nearest absolute deadline is dequeued and selected to be executed. Since both WCRQ are using fixed priority, the access to the job with the highest priority and the comparison can be done in constant time.

To support constant priority assigning at each job release, the kernel needs to be able to keep track of the number of releases made by each task, both periodic and sporadic. The same fixed priority queue structure used in FIASCO, with the modification of using bitmap to allow dequeue to be executed in constant time, can be used.

7 Discussion

Thread preemptions is one reason why EDF may have lower overhead than RM [3]. The idea is that EDF will experience less preemptions since the release of a thread with a shorter period than the currently executing one will not necessarily need to be executed right away. The benchmark shows that running threads under EDF decreases the number of schedule calls compared to running the same threads under RM. Figure 4 shows a decrease in schedule calls the higher the utilization gets which consist of the result shown in "Rate Montonic vs. EDF: Judgement Day". At around 90% utilization, a 34% decrease in schedule calls for EDF compared to RM could be seen.
However, this doesn’t mean that EDF performs better than RM. Figure 11 shows the improvements in dequeue time for EDF over RM. It can be seen that for more threads, the dequeue time becomes longer for EDF compared to RM. RM performs better than EDF with 190 threads at 90% utilization. One important thing to remember is that the actual calls to either enqueue or dequeue are decreased significantly in EDF and it can give a false view of the CPU time spent in queue. For example, EDF with 190 threads and 90% utilization has 40% less calls to enqueue/dequeue compared to RM. Still RM spends less time in the dequeue than EDF. However, the time spent in dequeue for RM is still surprisingly high with lower utilization rate.

Since the dequeue operation in fixed priority has to look for next highest priority by iterating down the priority lists, an improvement from $O(n)$, where $n$ is the number of priorities, to $O(1)$ could be made with the use of bit map and hash function. This is one of the main motivations to create a fixed priority that mimics EDF scheduling. Since the queue operations could be done in constant time and because it mimics EDF scheduling, significantly less preemptions occur compared to using fixed priority with RM.

For the proposed approach to mimic EDF using fixed priority, described in Sec. 6.2.2, the main overhead is coming from creating the WCRQ for sporadic tasks. The queue for the periodic task only has to be created once unless a new task is added to the task set and therefore only an initial start-up period is needed. One benefit for using the WCRQ approach over the FPP is when the sporadic task is released in bursts or a large number of tasks is present. In FPP, for a large amount of tasks that needs priority promotions and being released in bursts a timeout is required for each of the active jobs. During the time
when jobs that came in bursts have their priority promotion, the system that is running FPP may experience heavy performance and responsiveness issues if one-shot timer is used. Even if a periodical timer is used, the jobs still need to be dequeued and enqueued into the right priority queue. The WCRQ approach will give the bursts of a task same priority and scheduled as usual according to that priority and therefore avoid this issue.

Since FPP uses a schedulability test to only assign priority promotions when needed and uses RM otherwise, FPP loses the benefit of EDF in regards to preemptions. FPP may experience more overhead because more preemptions will occur if several threads are scheduled using RM. The benchmark for a regular EDF implementation shows a decrease in schedule calls by 12-34% and therefore using an approach to fully mimic EDF could further improve performance.

A possible issue for the WCRQ method in regards to the sporadic task queue is that the queue may get invalid quickly if there are many sporadic tasks in the task set. Increasing the amount of tasks will also increase the chance of the minimum absolute deadline difference between two potential jobs being very small. This will result in unpredictable jitters during the execution because a new WCRQ have to be created at high frequency. Having a dynamic horizon that chances in regards to the size of the sporadic task set may help to ease the problem.

Under extreme heavy load and big numbers of threads, both of the WCRQs may experience priority overflow where there are not enough priorities to give out to jobs. This is an issue that has to be solved in order to make it a valid option. For the periodical WCRQ, the simulation could also include a test to assure that no priority overflow will occur during the worst case execution of the hyperperiod. It might also be possible to perform this test whenever a new sporadic task is added to the WCRQ because the worst case release behaviour for sporadic tasks is that they act as periodical tasks.

Systems with very limited memory may not be suitable to use the WCRQ solution since it need to store the priority for each job release. The end-user need to be careful choosing the periods for the periodical tasks because the hyperperiod for that task set will determine how many priorities needs to be saved. For sporadic tasks the amount of stored priorities is limited by the horizon.

If the creation of the WCRQ for sporadic tasks is too expensive, one solution is to schedule all sporadic tasks with a heap queue as done in the EDF implementation described in this thesis. The same queue for scheduling periodic tasks with fixed priority is kept, making the schedule overhead to be done in constant time except for the initial overhead for the simulation when adding a new task.

8 Conclusion

The benchmark has shown that the EDF implementation for FIASCO do make significantly fewer calls to the scheduler than RM. The major overhead for EDF comes from the ready queue that runs at $O(\log(n))$ time complexity for dequeue and enqueue. An approach has been proposed to combine fixed priority and EDF scheduling to potentially take advantage of the constant enqueue/dequeue time complexity of fixed priority and the fewer preemptions of EDF scheduling.
9 Future Work

Implement and further design the WCRQ approach and evaluate the performance.
Make a comparison between the FPP solution to see if any additional performance could be made.

10 Known Implementation Issues

- Very rare cases of assertion happen during fixed priority. Same memory is deallocated twice causing the following assertion to fail:
  
  **Assertion Message:** 
  
  ".../obj/l4/x86/include/l4/re/util/counting_cap_alloc:184: 
  Assertion "!l_items[c].is_free()" failed."

- The return calls for a thread seems to delay the removal of the scheduling context. This is causing an issue for real-time threads, false deadline misses, since the reset of timeouts occurs in the deconstructor of the scheduling context. A work around is to convert the real-time threads to system threads before calling the return function.
References


