Dissertation for the Degree of Master of Science
Advisor: Professor Doo-Hyun Kim

Enhancing eCos with EDF Scheduling and Lock-Free Buffer

August 8, 2010

Konkuk University Graduate School
Department of Computer, Information and Communication Engineering
Nodir Kodirov

Enhancing eCos with EDF Scheduling and Lock-Free Buffer

THE SUBMITTED DISSERTATION IS PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE MASTER OF SCIENCE

August 8, 2010

Konkuk University Graduate School

Department of Computer, Information and Communication Engineering

Nodir Kodirov

This is to certify that the Master dissertation of Nodir Kodirov has been approved by examining committee for the requirements of dissertation of the Master of Science degree in the Department of Computer, Information and Communication Engineering, Graduate School of Konkuk University in the August 2010 graduation.

Committee Members

Chairman Chun-Hyon Chang

Member Hyun-Wook Jin

Member Doo-Hyun Kim

August 8, 2010

Konkuk University Graduate School

Table of Contents

Preface

This work is the result of my two-year Master program done in Embedded Computing Laboratory at Konkuk University.

My first word of thank goes to my supervisor, Professor Doo-Hyun Kim for his priceless advices, support and encouragement during all of my study at Konkuk University. Without his precious support, helpful guidelines on how to conduct an intensive research this work was impossible. I am more thankful to him for the projects he provided me a chance to participate and get an insightful knowledge and experience on research and development area.

I would also thank professors from the departments of Computer, Information and Communications, and Internet and Media. My special thanks go to professors Dugki Min, Jungkeun Park, Namseo Goo, Kyoungro Yoon and Christian Rinderknecht for their academic advices and comprehensive lectures during my coursework.

Also, I am very thankful to all member of our exciting lab, including senior members Dongwoon Jeon, Keunsoo Lee and junior members Junyeong Kim, Haeseong Yun, Kiho Cho, Seunghwa Song, Khoa and Bokhee Ryu for their close support and collaboration, which definitely helped me to succeed in my master study.

Finally, I would like to thank my parents and family, for their unconditional support, endless love and continuous encouragement in all circumstances.

Figure index

ABSTRACT

Enhancing eCos with EDF Scheduling and Lock-Free Buffer

Nodir Kodirov
Department of Computer, Information & Communication Engineering
Graduate School of Konkuk University

In this work, we address two issues at embedded system. They are thread scheduling algorithm and lock-free thread message communication mechanism at RTOS (Real-Time Operating System) kernel. Both of them have a direct relationship with system efficiency and an indirect relationship with stability through timeliness. We illustrate the need for and suitability of EDF (Earliest Deadline First) scheduling algorithm at *prototype* application. Our prototype application is designed based on computational characteristics of embedded application, carrying Real-Time computing of Small Unmanned Helicopter's OFP (Operational Flight Program). OFP runs on eCos (Embedded Configurable OS) RTOS on x86 architecture based board. Also, we show suitability of our lock-free message communication mechanism for our prototype application. Implementation approach for both solutions will be explained fully, accompanying with source code skeletons. We demonstrate our enhanced kernel performance based on less context switch needed, higher CPU utilization allowance and finer timeliness qualities via various scenarios of producer-consumer applications, where prototype application is made of one of them.

Keyword : Real-time scheduling, EDF, eCos, feasibility, operational flight software, UAV

Chapter Ⅰ. INTRODUCTION

Current technology trend is leading towards embedded computing. Most of our daily appliances, portable devices, industrial and automotive tools are armed with embedded systems. All of them are targeted to do a particular job. However, what they share is to produce a presumed output by given time and quality. Generally, we understand these criteria as being an *efficient*. Computing efficiency is pivotal for all systems and embedded systems are not exception. Efficiency of the system is a general term and there is a number of ways to achieve it. In this work we approach it from two grounds, where both have a direct relationship with an efficiency and an indirect relationship with stability through timeliness of the system. The first is to enhance system scheduling algorithm and the other is to provide lock-free thread message communication mechanism.

As an experimental application, we have a prototype of the OFP (Operational Flight Program) carrying Real-Time computing of Small Unmanned Helicopter. Our both prototype and real embedded application runs on top of the eCos (Embedded Configurable Operating System) Real-Time Operating System (RTOS) for x86 architecture based embedded board.

The first ground where we aim to have a higher OFP performance is to add EDF (Earliest Deadline First) scheduling algorithm to the eCos kernel. EDF (also known as Least Time to Go) is deadline based dynamic scheduling algorithm used in real-time OSs. Motive to implement EDF to our system based on the similarity between OFP's computational and EDF scheduling characteristics. Also, theoretical framework congested during the last four decades on real-time scheduling algorithms show EDF to be one of the best scheduling

policies for timely critical embedded applications. Based on these arguments we expect EDF to perform better than its alternatives, especially with given periodical nature of the OFP.

The second point we are going to hook the kernel is thread message communication domain. Once again, need to make this addition came from our practical application - OFP. It consist of the six threads to read and write data, and to execute control logic. In this scenario, one group of the threads plays role of the producer and others are consumer. Generally, several concurrent communication primitives, such as message-boxes, mutex or semaphores can be used to provide thread communication. However, problem with them is on their lock-based approach for the shared resource management. Instead, we are going to use Non-Blocking Buffer (NBB) to achieve the same. NBB is lock-free thread message communication mechanism. In each *state* NBB allows designer flexible retry-strategy that better suits real-time application characteristics under development. NBB can have one of several states, when buffer is full, consumer is in the middle of the reading, buffer is empty or producer is inserting. In all above cases, NBB designer is free to choose his/her own solution based on NBB status at the particular time. Once we provide lock-free thread communication mechanism for our OFP, we will have less scheduler locks; thus having more room for other the urgent computation to be executed earlier. In this way, we will have indirectly positive impact on timeliness of the OFP and our kernel in a whole.

Although we don't illustrate direct implementation and experimental results for our real OFP, we built our prototype application sharing most of the characteristics and computational nature to be as it is in real OFP. Thus, implementation and experimental results illustrated for our prototype application will confirm kernel enhancements to be true for our real OFP as well..

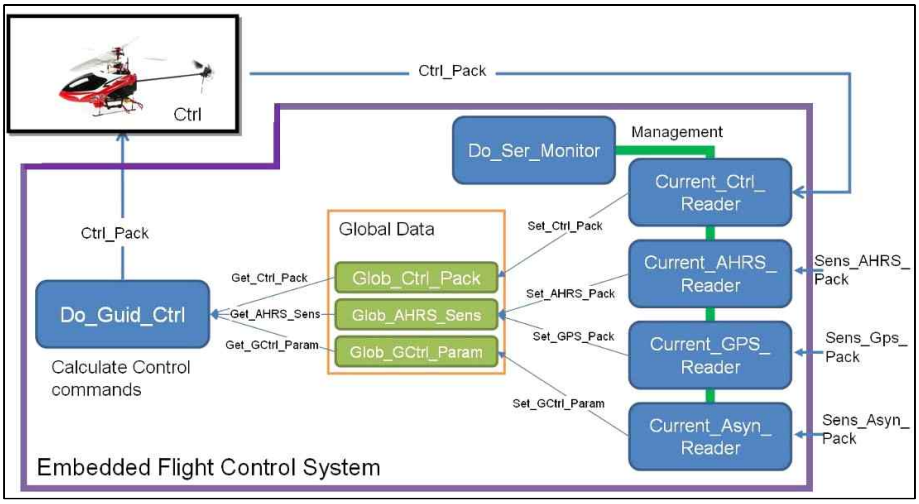
Rest of the work organized as follows. First, we will provide problem definition, explaining more about computational characteristics of UAV OFP, where our implementation is targeted to be utilized. Here, we will state the reason why we have followed this approach to make an improvements and what do we expect from our *eCos-EDF* and *eCos-NBB* implementations. Next, we will stop on related works addressed similar issues on real-time embedded systems. Before moving to the explanation of our approach and solution in detail, we will briefly explore eCos itself, its kernel and schedulers. This serves as a base for the all other coming sections. eCos-EDF and eCos-NBB implementation details will be fully illustrated, providing source code excerpts in the critical points. Also, we will introduce scheduling feasibility test support we have integrated into the eCos kernel space. In the penultimate section, we will demonstrate performance improvements introduced by both of our solutions. Lastly, we will conclude our work summarizing key points and stating future works.

Chapter Ⅱ. Problem definition

Ultimately, our solution is targetted to be applied to the flight control program of the small UAV, used on disaster response and recovery phase of disaster management system [15]. UAV is supposed to reach remote and/or hazardous places to take a moving and still pictures of the hot spot location, together with capability of having up to 20 kg payload (which could be first aid box). In order to get clear definition of the problem, we need to know computational characteristics of our OFP, which will be used to build prototype application. In the first sub-section of this chapter we will provide brief introduction to our OFP, emphasizing on its computational characteristics, threads and their responsibility. Based on introduced OFP characteristics, we will state the reasons why we selected EDF and NBB to be proper solution to provide better timeliness and performance for our OFP and eCos in a whole.

2.1 OFP characteristics

Operational Flight Program is a piece of software written in C programming language. It is used as a flight control program of the small Unmanned Aerial Vehicle (UAV). OFP consist of the six threads for reading and writing data and control logic.



<Figure 2-1> OFP thread interaction

The first thread interacting with Helicopter is Control logic thread, marked as the `Do_Guid_Cntrl` (Guidance and Control thread) in the Figure <2-1>. It is OFP's core thread running on 50Hz frequency. Its task is to calculate control signal based on input sensor data and control algorithm, and send them to the UAV's `Ctrl` (Control) module. The second one is `Do_Ser_Monitor` (Serial monitor thread) for polling asynchronous I/O ports. The other four threads are to read sensor data and store it into the global storage. These four reader threads manage Control Signal, Navigation, GPS, and Command message respectively. The `Current_Ctrl_Reader` (Control reader thread) thread receives control signal which is fed back from OFP. The `Current_AHRS_Reader` (Attitude Heading Reference System reader) thread is to receive `AHRS` sensor data sent on 100Hz frequency. Received data include current angle (roll, pitch, yaw), three-axis acceleration and angular velocity. Reading data from `Sens_AHRS_Pack`, `Current_AHRS_Reader` analyzes and stores into the Global data (`Glob_AHRS_Sens`). The `Current_GPS_Reader` (GPS reader) thread's role is to receive GPS sensor data sent on 10Hz frequency and save it into the Global data (`Glob_AHRS_Sens`). `Current_GPS_Reader` received data includes location values

(longitudinal, latitudinal and altitudinal), and velocity of three-axis. Finally, `Current_Asyn_Reader` (Asynchronous Data Reader thread) receives command data including waypoint, operational mode and UAV information, which are sent from Ground Control System (`GCS`). All received data by `Current_Asyn_Reader` are saved into the Global storage. Global data storage content will be used to calculate control signal for proper navigation and stability of the UAV.

2.2 Suitability for timeliness and performance

As it is mentioned above, we were targeting to have higher system performance and CPU utilization allowance to meet timeliness of our OFP and to meet future development needs. We came up with two ways to achieve our purpose. Both of ways roots to the appropriate features of our practical application - UAV OFP. Our first approach to enhance OFP performance is to add EDF scheduling algorithm to the eCos kernel. EDF is deadline based dynamic scheduling algorithm used in real-time OSs. Motive to implement EDF to our system is based on the similarity between OFP's computational and EDF scheduling characteristics. Our OFP consists of six threads, executing by the given period to read sensor data and output control commands. EDF is believed to be an optimal scheduling policy in preemptive uni-processor system with a set of threads executed by their deadlines [11]. Also, EDF scheduling upon uniform multiprocessors is robust with respect to both job execution requirements and processor computing capacity [6].

We expect deadline based common characteristics shared by both OFP and EDF, which is parallel execution relationship amongst sensor data read threads and their periodical execution to be a suitable match. This two features are expected to provide high performance for finer

timeliness. Also, theoretical frameworks congested during the last four decades on real-time scheduling algorithms confirm EDF to be one of the best scheduling policies for timely critical embedded applications. Based on all of these arguments, we expect EDF to perform better than its alternatives, especially with a given periodic nature of the OFP.

Usually, Fixed Priority algorithms are relatively in a wide use in industry. The reason for this is that fixed priority algorithms are simpler to implement on top of the commercial kernels. A detailed comparison between Rate Monotonic and EDF has been discussed by Buttazzo [3] under several perspectives. Despite their advantages, dynamic scheduling methods are not widely used in embedded real-time systems. During this thesis we show by using suitable kernel mechanisms for time representation and scheduling, EDF can be effectively used for enhance system utilization and achieve a timely execution of periodic tasks for the OFP.

The second approach we are going expand eCos kernel is thread message communication domain. As at the previous point, need to make this addition came from our OFP. It is consist of the six threads for reading and writing data, and to execute control logic. In this scenario, control logic and other five threads are in the sequential relationship, which makes room for lock-free NBB communication mechanism. Here, one group of the threads plays role of the producer and other consumer. These producers and consumer together manipulate global data to communicate with each other. Basically, eCos provides several primitives, such as message boxes, mutex or semaphores to provide thread communication mechanism. However, problem with them is their lock based nature for the shared resource management. Instead, we are going to use Non-Blocking Buffer (NBB) to achieve the same. NBB is lock-free thread message communication mechanism, which provides flexible retry-strategy selection for the designer to execute custom piece

of code in various NBB states. NBB can have one of several states, such as `BUFFER_FULL`, `BUFFER_FULL_BUT_CONSUMER_READING`, `BUFFER_EMPTY`, `BUFFER_EMPTY_BUT_PRODUCER_INSERTING` (Figure <4-14>) based on its status at particular time. Lock-free nature and retry strategy flexibility enables us to have a efficient thread message communication mechanism, thus offering us more responsive kernel with less scheduler locks. In this way we will improve timeliness of our OFP and eCos kernel in a whole.

Chapter Ⅲ. Related works

My exploration of real-time scheduling implementation to add EDF on eCos started at an early graduate study. It is almost two years. However, the first seminal work in this domain was published at 1973 by Liu and Layland [1], which is almost half a century ago. At their historic publication they defined and analyzed EDF and RM (Rate Monotonic or Fixed Priority) real-time scheduling algorithms. Since that vast amount of research material was produced, to examine and contrast schedulers by several aspects, some of them came up with practical implementations and results.

Lightweight EDF scheduling with deadline inheritance [8] is one of works appeared in a literature. Their approach is to make EDF scheduler targeted to be implemented in feather-light micro kernels. Generally, on their design they require and limit application thread context switch to be minimum, thus allowing application programmer to think of threads behavior to be run-to-completion. In this scenario one thread enters critical section blocking others and during execution no other thread can preempt it. Mutual exclusion of shared resources is granted at system level and programmer does not need to take care of synchronization: processes are simply not scheduled by the system when there is a potential resource conflict. However in our EDF implementation, we don't do any implicit assumptions and threads with higher "priority" are allowed to preempt executing ones.

Another work [7] discusses combined implementation of EDF and FP (Fixed Priority, similar to our MLQ scheduler) in Ada. They made good mix, having efficiency from EDF and predictability from FP. Although their approach and theoretical framework has a good foundation, they implementation is in Ada programming language, it is not implemented

as a standalone scheduling algorithm in RTOS kernel.

Our idea of the NBB is originated from the paper published by K.H (Kane) Kim et al. [2]. They illustrated lock-free mechanism for event message communication for distributed computing objects and other domains. We followed that concept to extend our eCos kernel with additional thread communication mechanism to achieve less locks at shared resources control, achieving more responsive kernel.

Generally, there is only one non-blocking event message exchange scheme that uses a circular buffer and appeared in literature [4]. NBB authors explain NBB to have the following advantages. First, a consumer thread can be designed to perform a retry strategy that better suits the specific characteristics of the real-time application under development. Secondly, its implementation does not require use of complicated atomically executing machine instructions other than simple integer write and integer read operations. Other proposed non-blocking schemes for exchanging event messages are based on the use of linked lists [5] and they incur higher overhead.

Paper [4] introduces Non-blocking FIFO queue for the multiprocessor systems. Basically, they address three points, where lock-based mechanisms fail to efficiently control shared data. They are:

- They avoid convoys and contention points (locks);
- They provide high fault tolerance (processor failures will never corrupt shared data object) and eliminates deadlock scenarios, where two or more tasks are waiting for locks held by others;
- They do not give priority inversion scenarios;

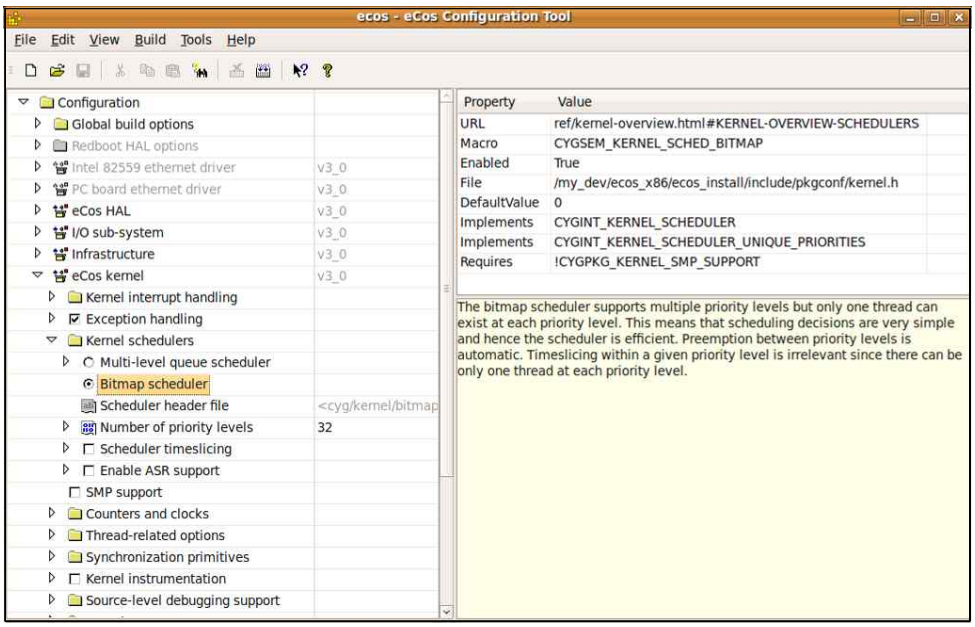
They introduce simple mechanism to overcome above three problems, by their non-blocking FIFO queue. Generally, their target apply domain differs from ours (they are addressing lock-free operations on shared data between multiprocessors, but we between threads in uni-processor systems). But, relevant and interesting part of this work is clear

illustration of non-blocking mechanism to outperform their lock-based counterparts. Another work [21] illustrates benefits of the non-blocking object sharing approaches in uni-processor systems and with the use of schedulers to bound interference of threads. However, their approach is simply retry-again, in case of failure to access shared data, while our NBB has advantage of providing NBB designer with custom retry-strategy when buffer is not available due to some reasons.

Chapter Ⅳ. Our approach and solution

In this section we will provide detailed explanation of our work. Before moving to the explanation of our approach and solution in detail, we will briefly explore eCos itself, its kernel and schedulers, which are prerequisite knowledge needed by all other coming chapters.

eCos is Open Source RTOS available under GNU General Public License [9]. Mainly it is known for its high customization and low memory footprint in target image [10]. eCos kernel is programmed in C/C++ language and kernel developers are provided with handy resources to make their own modifications and/or customization. Also, it has compatibility layers and APIs for POSIX and μITRON, also support for dozens of platforms and architectures [9, 10].



<Figure 4-1> GUI eCosConfigTool

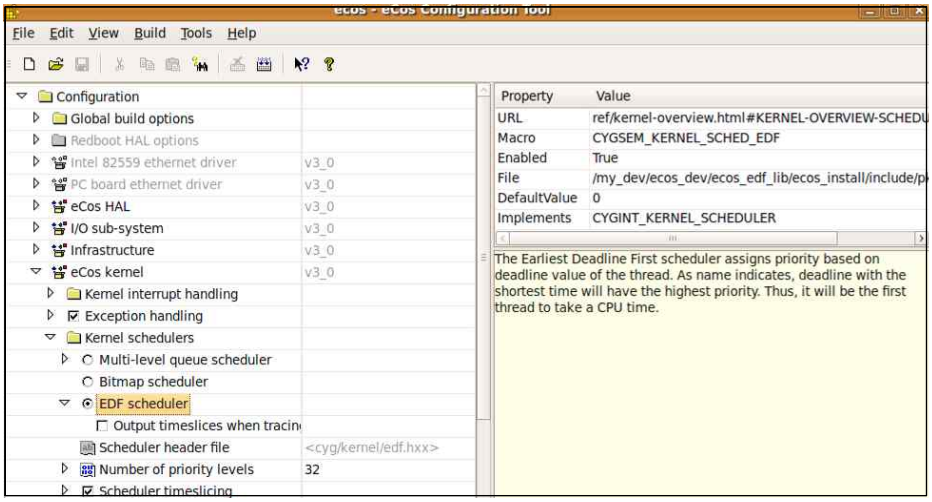
eCos designed to be highly customizable to meet application requirements and hardware needs. It is achieved through GUI tool called as eCosConfigTool. It enables developers to have a kernel with specific

features needed by a particular application. Currently, the latest stable release eCos 3.0 by March 2009 includes more than several hundred configuration points [10]. Snapshot of eCosConfigTool is illustrated in the Figure <4-1>. Here, application developer is provided with class configuration points, grouped under similar characteristics. Figure above shows options of the eCos HAL (Hardware Abstraction Layer), I/O sub-system, Infrastructures, kernel and so on. In our work, we are going to add support for the EDF scheduler under eCos kernel schedulers and NBB message communication mechanism under kernel synchronization primitives option groups.

eCos 3.0 release includes two completed and one ongoing kernel scheduler. Full supported algorithms are Bitmap and Multi-Level Queue, and ongoing one is implementation of Lottery scheduler by eCos developers. Starting from an early release eCos came with a static priority scheduling algorithm, shown as a “Bitmap” in the Figure <4-1>. Bitmap is static scheduling policy, where application programmer is supposed to provide unique priority for each application thread. As eCos keeps all extant threads as a double linked list and uses 32-bit integer variable to easily locate highest priority one, application programmer is limited to have maximum 32 threads. Moreover, there are two system supplied `Main` and `Idle` threads, which reserve priority 8 and 31 by default.

Limitation to the number of application threads lead to the addition of the Multi-Level Queue (MLQ) scheduler. In this scheduler application programmer can have several threads (32 by default) at the same priority. Currently, thread execution order can not be pre-determined if there are two or more threads at the same priority competing for CPU time. Again, in the source code implementation, kernel uses 32-bit integer variable to easily locate the highest priority thread, which makes us to have up to 32 priorities.

During our research we added EDF scheduler keeping standard way to add new scheduler support for eCos kernel [10]. Now, application programmer is free to choose one of three schedulers, including EDF from GUI eCosConfigTool based on its application requirements, Figure <4-2>.



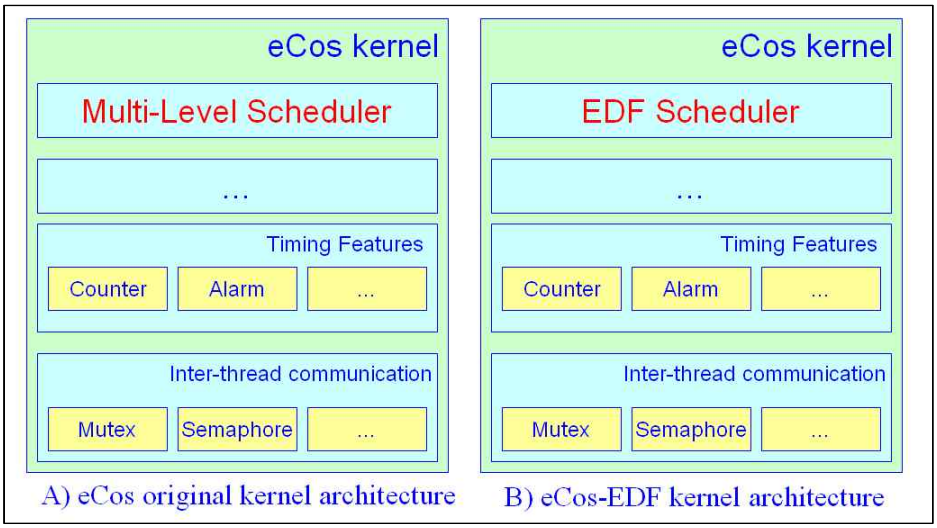
<Figure 4-2> EDF scheduler in eCosConfigTool

Generally, eCosConfigTool defines each configuration option as a unique `Macro` in the specific files manipulated by it. In our case, EDF scheduler macro is `CYGSEM_KERNEL_SCHED_EDF` as it is shown in the left side of the figure above. Once application programmer enables this option, it will have an appropriate value in the header file. Thus, respective EDF specific code will be included into the target image.

4.1 Kernel extension design approach

There are various EDF implementation approaches appeared in a literature [7, 8]. Although, implementation approach may differ for various RTOSs, but in general, EDF is expected to have a higher CPU utilization and efficient scheduling. In our implementation we put EDF scheduler backbone down to the kernel data structure level and started to build

additional supports [12]. In this way we have used MLQ scheduler specific codes for scheduler-independent kernel primitives, such as timing (clock, alarm, counter and etc.) and inter-thread communication (mutex, semaphore, message box and etc.) mechanisms. Our attitude was to make the EDF specific modifications wherever it is required by EDF concept. Figure <4-3> illustrates the approach we have followed for our implementation.



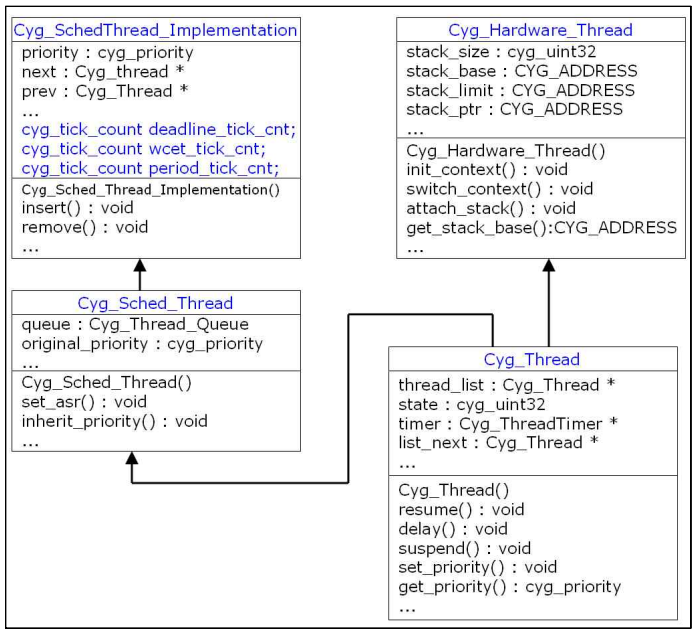
<Figure 4-3> eCos-EDF kernel

As it was mentioned before, for EDF development we used latest stable version of eCos 3.0. In addition to Multi-Level Queue (MLQ) and Bitmap scheduler, we extended eCos kernel with EDF scheduling policy. eCos kernel source code exploration enabled us to see kernel object and their relationship. Based on these knowledge, we added EDF specific data structures as it is illustrated in the Figure <4-4>. We added EDF specific:

```
cyg_tick_count deadline_tick_cnt;
cyg_tick_count wcet_tick_cnt;
cyg_tick_count period_tick_cnt;
```

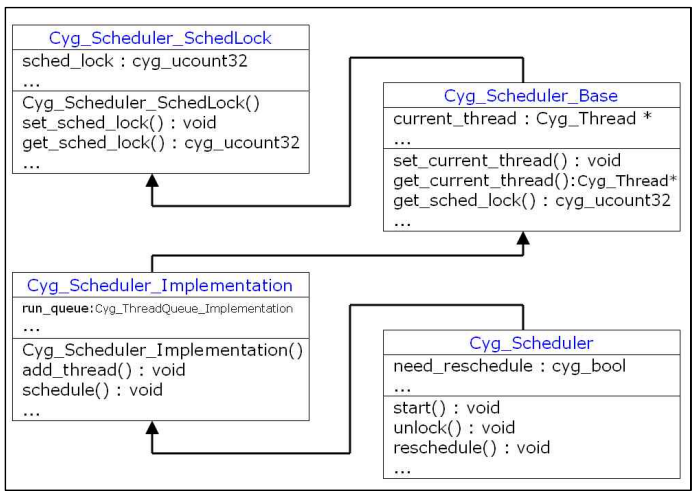
fields to our each `Cyg_Thread` via its parent class `Cyg_SchedThread_`

`Implementation` class. As each eCos scheduler has its own `Cyg_SchedThread_Implementation` class implementation, we made our in EDF header and source files (Figure <4-9>).



<Figure 4-4> eCos-EDF classes and their relationship

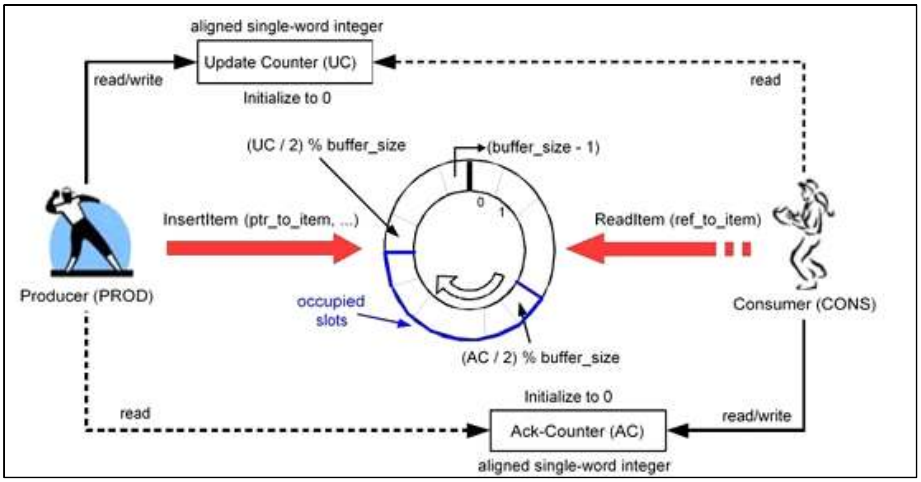
Also, we put EDF scheduling logic in `schedule()` member function of `Cyg_Scheduler_Implementation` class, Figure <4-5>. More details will be explained in the next section.



<Figure 4-5> eCos-EDF scheduler logic code

Now, we will move to the explanation of design approach for NBB. Generally, all Operating Systems provide several thread communication mechanisms for the various scenarios. The most widely used are message boxes, mutex and semaphores. However, problem with them is their lock based approach for the shared resource management. All of them follow lock-based technique to control shared data. As it was explained in the section about OFP characteristics, it is consists of six threads, communicating through global buffer. Current OFP source code uses mutex for thread communication, which makes our whole code to be lock dependant, degrading system performance and timeliness. Instead, we are going to use Non-Blocking Buffer (NBB) for the thread communication. NBB is lock-free thread message communication mechanism, which provides NBB designer flexible retry-strategy selection capability to execute custom piece of code that better suits the specific characteristics of real-time application under development. Once we will be able to provide lock-free thread communication mechanism for our OFP, we will have less scheduler locks, having more room for other the urgent computation to be executed earlier. In this way, we will have indirectly positive impact on timeliness of the OFP and our kernel in general.

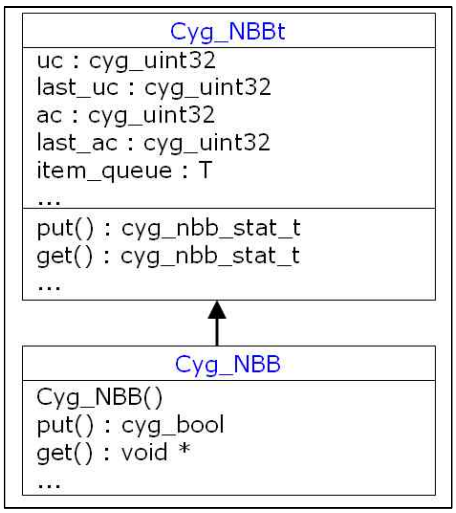
The NBB is a circular FIFO queue that facilitates the communication of event messages from a single producer thread (PROD) to a single consumer thread (CONS) without causing any party to experience blocking [2]. High-level view of the system is illustrated in the Figure <4-6> (used from [2]).



<Figure 4-6> High-level view of the NBB

As it can be seen from the figure above, NBB uses two integer variables, which are `Update Counter` and `Acknowledgement Counter` to provide control of the shared buffer items. There is a third variable `Recycle Counter` to control the items already consumed from the buffer, so called `Defunct Item`. The underlying mechanism for NBB to execute critical operations without locking is achieved through those three variables. Variables should be defined in such a way, where their read or write operations are done via single processor instruction. Thus, we will not have an uncompleted operation left after each processor instructions. Generally, the purpose of the lock is to maintain data consistency at runtime. But in NBB, as we don't have any instructions executed in two steps (no operation is executed in the middle of single integer read and write operation) we will have totally consistent data.

With our kernel source code knowledge, we designed NBB to have template and actual object class. Abovementioned four critical single instruction integer class member variables were put into the `Cyg_NBBt` template class (Figure <4-7>). Required `put()` and `get()` member functions were implemented at `Cyg_NBB` class. NBB source and header files contain Template and Class codes, Figure <4-9>.



<Figure 4-7> NBB Template and Class

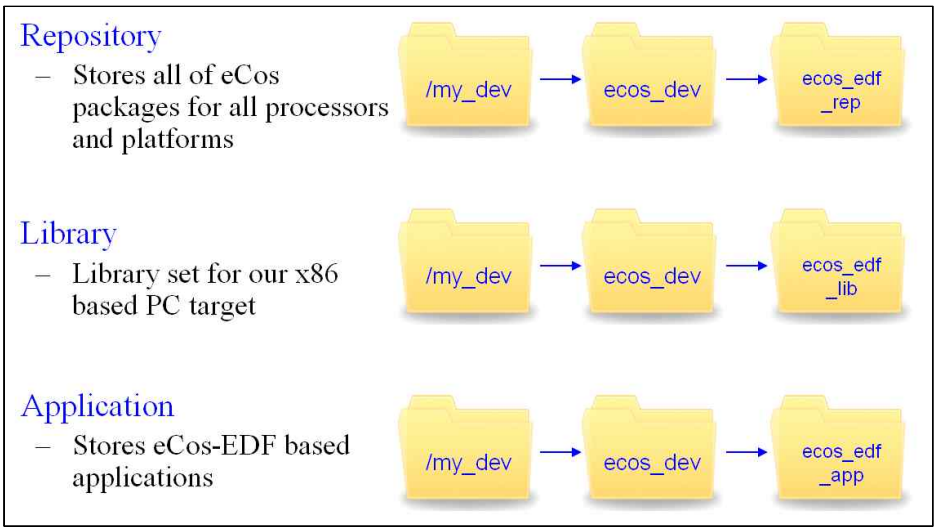
The approach we followed to add NBB support to our eCos kernel was standard way of kernel extension [10]. First we added appropriate configuration options in eCosConfigTool and further we built necessary support in eCos kernel itself. Implementation details are illustrated in the next sub-chapter.

4.2 Implementation details

In this chapter we will illustrate our implementation in a comprehensive way. We will start from EDF implementation details and based on skills gained during this step, we will be able to make ease transition to the NBB source code details.

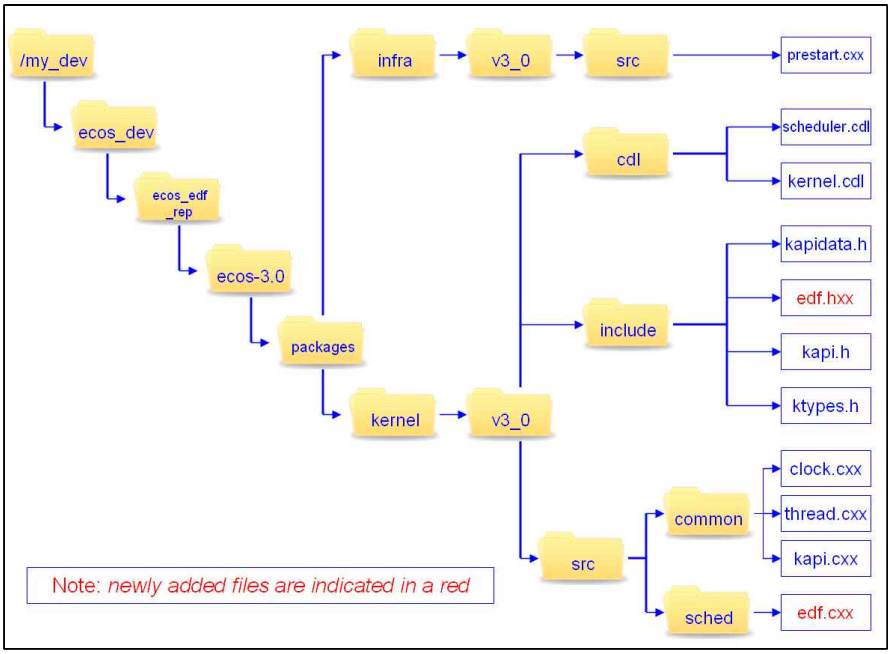
Before going to deep in details, we will briefly illustrate our development environment. Our development OS is Linux UbuntuTM 9.4 distribution [13] and SciTe text editor to edit C/C++ source codes [14]. When user downloads eCos 3.0, it comes with all packages and libraries for all target platforms and architectures. It is called as "eCos repository". In order to make our target specific library, we should execute `ecosconfig new pc` command to the Ubuntu Terminal and we

will have desired eCos library. eCos repository, target specific library and EDF application development folder structure is depicted in the Figure <4-8>.



<Figure 4-8> eCos-EDF development folder structure

During eCos-EDF implementation we edited several configuration files and source files, and added new files by the need. All of them are illustrated in the Figure <4-9>. As it can be seen from the figure below, there are only one header file and source file was added to the eCos repository. All other files were only modified according to EDF requirements. Other than header and source files, there are two files with "cdl" extension, which are responsible to control eCosConfigTool options. "cdl" stands for Component Description Language. We added EDF scheduler option to the `scheduler.cdl` file, making one additional note in `kernel.cdl` file to add EDF specific option at the compile time of our kernel.



<Figure 4-9> Modified and added EDF specific files

From now on we can start to dig deep into EDF specific code implementation. As we mentioned above, we put EDF scheduler backbones to the kernel data structures. Basically, EDF requires thread to have three properties, *deadline*, *worst case execution time* (wcet) and *period* [11, 12]. The EDF requires each thread to have its deadline value represented in RTOS specific time units. In the case of eCos it is clock tick generated by the hardware (HW) clock interrupt. With Hardware Abstraction Layer (HAL) eCos provides 100 Hz tick frequency for all platforms. As our OFP application does not require any higher clock frequency, we will keep this unchanged. Application programmer is free to have a custom clock frequency via system supplied resolution set mechanism. Basically, eCos clock tick count is used as a primary synchronization unit for the thread alarms, delays and suspends. Thus, in order to keep interoperability with other kernel primitives, we also designed EDF fundamental data structure to be in target specific clock ticks:

```
struct cyg_sched_edf_info_t {
  cyg_tick_count deadline_tick_cnt;
  cyg_tick_count wcet_tick_cnt;
  cyg_tick_count period_tick_cnt;
}
```

This C structure with three properties was added as a kernel data structure, to the `kapi.h` header file, where `cyg_tick_count` is wrapper type for `long int` of C language. We added the same structure to the thread control block (TCB) of eCos, where <cy4>. In this way each thread will have thread's `deadline`, `wcet` and `period` as it is required by the EDF.

```
314  #elif defined(CYGSEM_KERNEL_SCHED_EDF)
315  # define CYG_SCHEDTHREAD_SCHEDIMP_MEMBERS                    \
316      cyg_thread *next;                                        \
317      cyg_thread *prev;                                        \
318      cyg_priority_t     priority;          /* current thread priority */ \
319      struct cyg_sched_edf_info_t *edf_info;                   \
320      cyg_tick_count deadline_tick_cnt;                        \
321      cyg_tick_count wcet_tick_cnt;                            \
322      cyg_tick_count period_tick_cnt;                          \
323      CYG_SCHEDTHREAD_CPU_MEMBER                               \
324      CYG_SCHEDTHREAD_TIMESLICE_MEMBER                         \
325      CYG_SCHEDTHREAD_TIMESLICE_ENABLED_MEMBER
326  #else
327  # error Undefined scheduler type
328  #endif
```

<Figure 4-10> EDF specific properties in TCB

In the Figure <4-10> it is illustrated EDF specific codes (316-325 lines) to be inside its macro `CYGSEM_KERNEL_SCHED_EDF`. The macro is controlled by eCosConfigTool. Lines from 320 to 322 illustrate EDF specific thread options in clock ticks. Application programmer is supposed to provide accurate values for the thread's {`deadline, wcet, period`} triple in milliseconds during its creation time. All these three are supplied into the kernel's `cyg_thread_create()` thread creation method and put into the TCB as a thread's native property (Figure <4-12>).

In eCos all of the threads are in the "suspended" state once they are created. Thus, application programmer needs to explicitly call

`cyg_thread_resume()` function to make them ready and start execution. Based on this split mechanism to create thread and make it available for the scheduler, we made additional `edf_prep()` function for the application programmer to make a call to it between thread creation and resume. This is done through call to the `cyg_prestart()` function in the `prestart.cxx` source file inside infra package illustrated in the Figure <4-9>. This function is middle execution step provided by the eCos for the application programmer to initialize custom packages if there is any. If there is no, this function simple passes execution to the next function, without doing anything inside. Figure <4-11> illustrates modified version of this function. Call to the `edf_init()` (which redirects call to `edf_prep()`) EDF preparation function is shown in line #90.

```
77  void
78  cyg_prestart( void )
79  {
80      CYG_REPORT_FUNCTION();
81      CYG_REPORT_FUNCARGVOID();
82
83      CYG_TRACE0( true, "This is the system default cyg_prestart()" );
84
85      CYG_EMPTY_STATEMENT; // don't let it complain about doing nothing
86
87
88  #ifdef CYGSEM_KERNEL_SCHED_EDF
89          // edf_init() initializes clock frequency and basic priority for EDF threads
90          Cyg_SchedThread_Implementation::edf_init();
91  #endif
92
```

<Figure 4-11> Call to the EDF preparation function

This call-in-the-middle is the very right place to execute special piece of code, to achieve two major objectives. The first is to interpret user provided millisecond values into the platform specific clock-tick units in TCB. Hence, we will get platform-independent timing mechanism. The second objective is to make scheduling feasibility test, which is explained in the coming section. Below is the source code excerpt from `edf_prep()` to achieve the first objective:

```
650   void Cyg_SchedThread_Implementation::edf_prep()
651   {
652       //CYG_TRACE1(1, "from edf_schedule()", true);
653       Cyg_Thread *thread = Cyg_Thread::get_list_head(); // get_current_thread();
654       cyg_count32 sum = 0;
655
656       while ( thread ) {
657
658           if ((thread->priority == CYG_THREAD_MIN_PRIORITY) ||
659               (thread->priority == CYGNUM_LIBC_MAIN_THREAD_PRIORITY))
660           {
661           } else {
662               thread->priority = (cyg_priority) edf_priority();
663
664               thread->deadline_tick_cnt = thread->edf_info->deadline/ms_per_system_tick;
665               thread->wcet_tick_cnt = thread->edf_info->wcet/ms_per_system_tick;
666               thread->period_tick_cnt = thread->edf_info->period/ms_per_system_tick;
667
668               sum += (100*thread->edf_info->wcet)/thread->edf_info->period;
669
670           }
671           thread = thread->get_list_next();
672       }
673       if (sum >= 100)
674       {
675           CYG_TRACE1(1, "CPU usage is  %d %", sum);
676           CYG_ASSERT( false, "Non-feasible schedule, pls correct thread's wcet/period.");
677       }
678   }
```

<Figure 4-12> Scheduling feasibility test

Lines from 664 to 666 are converting millisecond values to the system specific clock ticks. Here, `thread->edf_info` contains user supplied values in milliseconds. Division to the current board specific milliseconds per system HW clock tick will give us their tick represented values. Lines 658-660 used to eliminate system default (`Idle` and `Main`) threads to be assigned priority based on EDF specific values (`deadline, wcet, period`). Because only these two threads are considered to be the system thread and they have eCosConfigTool specified priority values in the macro named `CYG_THREAD_MIN_PRIORITY` and `CYGNUM_LIBC_MAIN_THREAD_PRIORITY` respectively. `if` condition used to isolate EDF specific scheduling scheme from system owned threads.

After the initial execution, we will keep decreasing thread's `deadline_tick_cnt` value for the all TCBs at each clock tick. As scheduler returns the thread with the smallest deadline value, we will have purely EDF scheduling sequence. Once thread finishes execution, we will remove it from the ready list and whenever it becomes ready, we

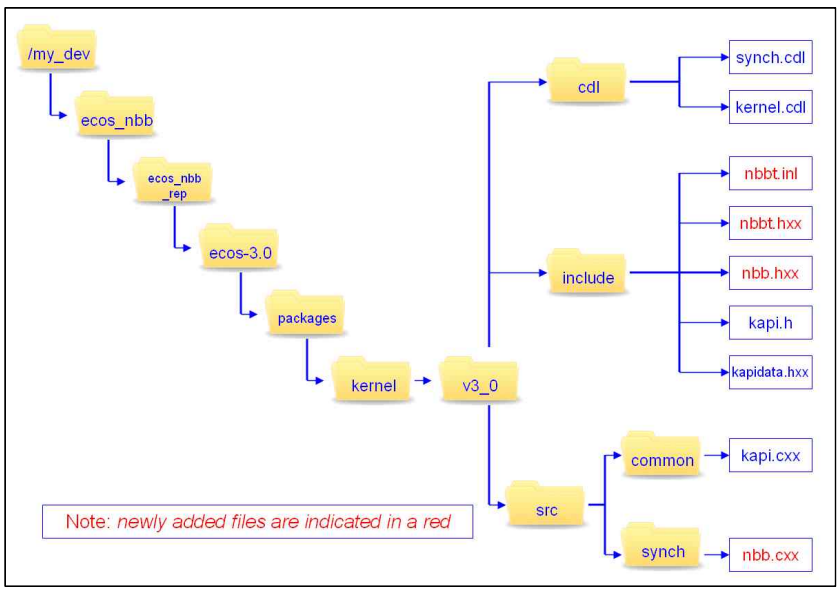
will assign new deadline value for `deadline_tick_cnt` as we did in `edf_prep()`. But, this time we do this only for `deadline_tick_cnt` not for the `wcet_tick_cnt` and `period_tick_cnt`, Figure <4-13>.

```
607  - #ifdef CYGSEM_KERNEL_SCHED_EDF
608                if ((this->priority == CYG_THREAD_MIN_PRIORITY) ||
609                    (this->priority == Cyg_Thread::get_main_thread_priority()))
610  -             {
611                } else {
612                    this->deadline_tick_cnt = this->period_tick_cnt - this->wcet_tick_cnt;
613                }
614   #endif
```

<Figure 4-13> Newly joined ready thread

Code in the figure above is located in the `resume()` function of the `Cyg_Thread` class. The reason why it is put there is that, `resume()` is the only function which makes thread into the ready state and appends it to the scheduling queue. As we put new value of the deadline just before thread joins the queue, it matches exactly what we want to achieve. Code in the lines from 608 to 610 does the same as it was explained for the Figure <4-10>. Code in the line 612 shows new `deadline` value of the thread, based on its `period` and `wcet`.

Now, we have finished explanation of eCos-EDF and we can make slight transition to NBB implementation path.



<Figure 4-14> NBB development folder structure

As it was in the case of EDF, for NBB we made underlying development folder structure first. It is illustrated in the Figure <4-14>. We created totally independent and new development settings (than we had for eCos-EDF) for eCos-NBB library and eCos-NBB repository. However, backbone folder structure is similar as it was for eCos-EDF, Figure <4-9>. As it can be seen from the Figure <4-14>, together with modification of existing kernel files, we created four more custom source and header files in appropriate folders. Here, `nbbt` stands for *NBB Template,* serves as a base template class when creating NBB instances. This is derived from eCos kernel synchronization primitive creating convention [10] and it is to provide flexibility for application programmer to define type of the items to be controlled by or stored at NBB circular buffer (Figure <4-19>). Generally, existing synchronization primitives, such as message box can be used to exchange all data types, including `int,ferng, double, char, void*` and so on. Via template class, we can provide application programmer to select data type that better suits application needs under development. For this reason application programmer should create NBB class object specifying data type in the template type (Figure <4-19>). In our NBB implementation, scenario is much different. NBB items are exchanges as a `void *` pointers, which can point to the any data type, and circular buffer slot is considered to store pointer address values only.

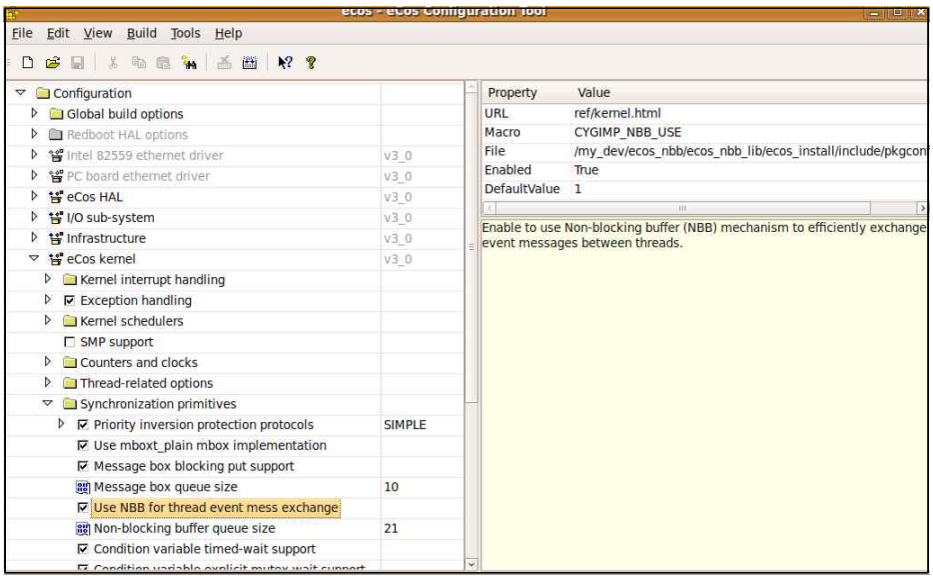
```
491  struct cyg_nbb
492  {
493      cyg_uint32 uc; // Update counter
494      cyg_uint32 last_uc; // Last update counter
495
496      cyg_uint32 ac; // Acknowledge counter
497      cyg_uint32 last_ac; // Last acknowledge counter
498
499      void *          itemqueue[ CYGNUM_KERNEL_SYNCH_NBB_QUEUE_SIZE ];
500  };
```

<Figure 4-15> NBB properties in eCos kernel

Figure <4-15> illustrates items to be stored in a variable called `itemqueue`, which is an array with `CYGNUM_KERNEL_SYNCH_NBB_QUEUE`

_SIZE number of void* elements. The macro name and other variables are explained in the consequent paragraphs.

As we did for the EDF implementation, for NBB implementation first we added NBB specific GUI configuration options in eCosConfigTool. We added NBB control option points next to the other shared data management tools, such as mutex, semaphore and message box. In eCos they are called as a "*kernel synchronization primitives*" and they are located under respective option group in eCosConfigTool, Figure <4-16>.



<Figure 4-16> NBB in eCosConfigTool

As it can be see from the figure above, there is two configuration points in eCosConfigTool. First one called as "*Use NBB for thread event mess exchange*", which is used to either enable or disable NBB specific code in kernel. Once we disable this option, resulting eCos library will not contain NBB specific code and application programmer will not be able to use any NBB specific API. The second NBB specific option is to used to set number of slots in the NBB circular buffer, in the other words in will set the value for CYGNUM_KERNEL_SYNCH_NBB_QUEUE_SIZE macro, thus defining size of the array in Figure <4-15>. In the

Figure <4-16> this option is called as "*Non-blocking buffer queue size*" and has a value equal to 21.

In the Figure <4-15> there are four variables defined as `uc`, `last_uc`, `ac`, and `last_ac` to be used for update counter, last value of update counter, acknowledgement counter and last value of the acknowledgement counter respectively. As we have mentioned early in this chapter and as it is explained in [2], in order to keep lock-free mechanism, operations done on these variables should be single instruction processor operation. In our implementation, we used `cyg_uint32` kernel specified data type, which is made of 32-bit unsigned integer. As we use 32-bit processor in our embedded board, all operations on this data type is guaranteed to be single processor instruction. In this way we achieve atomicity of our NBB specific operations, as it is required by NBB concept [2].

One of the main advantages of NBB is flexible retry-strategy NBB designer is provided. Designer is free to set custom action that better suits the specific characteristics of real-time application under development in various NBB states. NBB can have one of several states, when buffer is full, consumer in the middle of the having, buffer is empty or producer is inserting. In all abovvarises, NBB designer can define various solutions based on NBB status at the particular time. In our implementation, this feature is provided via of addition of special enumeration in the kernel data structure, Figure <4-17>.

```
313   typedef enum
314 - {
315       BUFFER_FULL,
316       BUFFER_FULL_BUT_CONSUMER_READING,
317       INSERT_DONE,
318       BUFFER_EMPTY,
319       BUFFER_EMPTY_BUT_PRODUCER_INSERTING,
320       READ_DONE
321   } cyg_nbb_stat_t;
```

<Figure 4-17> NBB states enumeration

`cyg_nbb_stat_t` enumeration type is return type for our core NBB

writer and reader functions in the template. Values from the line 315 to 317 could be return value for writer and the other three for reader function. Based on NBB status values our *retry-strategy-selection* (explained in the next paragraph) functions will execute custom piece of code.

In our solution, we have provided three layer of control for NBB writer and reader functions. The first two for both writer and reader are to do respective task inside kernel space. The other one is NBB API for application programmer to make use of NBB. For the writer, kernel space functions are `put()` at `Cyg_NBBt` template class (Figure <4-18>) and `Cyg_NBB` retry-strategy-selection class (Figure <4-19>). For the reader, it is `get()` functions at respective classes as in the above case.

```
70  template <class T>
71  class Cyg_NBBt
72  {
73  public:
74
75      cyg_nbb_stat_t   put( const T ptr_to_item, T &def_item );   // put an item;
76
77      cyg_nbb_stat_t   get( T &ritem );       // get an item; wait if none
78
```

<Figure 4-18> Writer and reader functions definitions

As we can see from the figure above, `put()` function takes two arguments, first is `ptr_to_item` pointer to item to be put in the next available slot in circular buffer and the second is defunct item `def_item`, which is the item already consumed by reader. Although, NBB provides mechanism to manage defunct items, in our application we don't have a need for this feature. Thus, in order to be fully compatible with NBB mechanism, we added this feature to our core template function. However, our current implementation just ignores returned defunct items. Figure <4-19> illustrates definition of retry-strategy-selection `put()` function in `Cyg_NBB` class. It takes only a pointer to the item going to be inserted in the next available slot and returns `booleain` triable to the application. The reason why we called `Cyg_NBB` class

member functions as a retry-strategy-selection one is, here as an NBB designet ae can defineetryticular strategy to be done in specificcatio n. Te (Figure <4-17>). In our initial solution, we followed simple way. We return `true` only if `Cyg_NBBt` core `put()` function returns `INSERT_DONE`, while in other two cases we return false to the application programmer. As our core template function operates purely by NBB concept (returning specific enumeration value at each NBB states), we can easily adapt to future development needs, modifying retry-strategy-selection function (`Cyg_NBB`'s `put()`) accordingly.

NBB reader is defined as `get()` function in `Cyg_NBBt` core and `Cyg_NBB` retry-strategy-selection class. As in the case of NBB writer, core `get()` function (line 77, Figure <4-19>) is implemented based on pure NBB concept. It returns one of `cyg_nbb_stat_t` enumeration values depending on item availability at NBB circular buffer, in the address of variable supplied at its argument list. This core function is called by our second layer NBB design (so called retry-strategy-selection) function, illustrated at line 81 of Figure <4-19>.

```
74  class Cyg_NBB
75  - {
76  public:
77      Cyg_NBBt<void *> m;
78
79      cyg_bool put( void *ptr_to_item ); // put pointer value to the slot
80
81      void * get( cyg_bool & stat ); // return pointer to the item
```

<Figure 4-19> NBB retry-strategy-selection functions

Depending on the return value of the core function, we are able to define our design. As it is illustrated in the above figure, `get()` function of the `Cyg_NBB` class returns address value of the item and `boelean` variable in its supplied argument variable address. Once again, as it is our initial implementation stage, we have follwed simple approach at retry-strategy of the NBB reader in `Cyg_NBB` class, Figure <4-19>. It returns pointer to the next available item, indicating its success or failure in the address of the supplied `boolean` variable. We return `true` in case

of result of core `get()` function is READ_DONE (Figure <4-17>), in both of the other two enumeration values we will return `false`. This boolean value is provided for the third layer function, which considered as an NBB API (explained in the next paragraph). It is worth to explain code line 77 in the Figure <4-19>. There we create an instance of NBB template to be generic pointer type, which indicates slots of the circular buffer to hold pointer to any data type.

We created several API to enable NBB usage by application programmer. Generally, eCos API is located in `kapi.cxx` source file, under `common` folder (Figure <4-14>). We followed the same design and appended NBB API in the same file. Currently, NBB API consists of three functions to create, read and write NBB items (Figure <4-20>). Once there is a need for additional API to make wider use of NBB, we can easily append them here.

```
921     externC void cyg_nbb_create(
922         cyg_handle_t      *handle,
923         cyg_nbb           *nbb
924     ) __THROW
925   + {
934
935     externC cyg_bool cyg_nbb_put(cyg_handle_t nbb, void *item) __THROW
936   + {
945
946     externC void * cyg_nbb_get( cyg_handle_t nbb ) __THROW
947   + {
970
```

<Figure 4-20> NBB API

`cyg_nbb_create()` gets to arguments, first is handle to the created NBB, which is returned to the application programmer to perform write and read operations later. The second argument is memory space provided to this NBB. In order to insert item to the NBB circular buffer, user provides NBB handle and pointer to the item to be inserted. Once write is success `cyg_nbb_put()` will returns `true`, if not `false`. Line 946 at Figure <4-20> is to perform read operation from NBB. For that user gives handle to the NBB, where item will be read from. If item is available and we could read it successfully, this function will return

pointer to the item, in other case it will return `NULL`.

In this point we finished implementation details to add EDF and NBB in eCos kernel. Interested reader can request author for more comprehensive explanation.

4.3 Scheduling feasibility test

In Real-time embedded computing systems, scheduling feasibility is considered to be one of the minor areas of the research [17-20]. There is no general approach to make feasibility test for all systems. For example, it will be different for fixed-priority, dynamic-priority scheduling environment or preemptive or non-preemptive kernel [16]. We found research [8] to be very similar with ours. They illustrate theoretical approach to be practical with a given properties of user supplied thread information. However, in our work we made practical implementation and testing via eCos RTOS kernel.

We have done simple mechanism to make scheduling feasibility test. Here, user is supposed to supply thread specific information at the time of thread creation. Generally, as we explained in the EDF implementation chapter, there are three EDF specific properties supplied thread creation time. They are `deadline_tick_cnt`, `wcet_tick_cnt` and `period_tick_cnt`, to indicate value of deadline, WCET and execution frequency of the thread respectively. However, only last two properties provided by the application programmer used to make the schedulability decision. If application fails to pass the test, kernel rejects it to run after prompting with an appropriate message about non-feasible schedule.

Simple way to make feasibility test is to check CPU utilization. It is obvious that utilization can not exceed 100%. It is made easy with EDF specific thread properties. It can be illustrated with following calculations.

$$T_{p_1}^{w_1} = \frac{T_1^{wcet}}{T_1^{period}} \qquad T_{p_i}^{w_i} = \frac{T_i^{wcet}}{T_i^{period}}$$

$$T_{p_1}^{w_1} + T_{p_2}^{w_2} + \ldots + T_{p_n}^{w_n} = U$$

$$U = \sum_{i=1}^{n} T_{p_i}^{w_i} \leq 1$$

In this equation `T` represents thread and `U` processor utilization. Theoretically EDF can have up to 100% CPU utilization [11]. Based on this assumption, we take division of `wcet_tick_cnt` over `period_tick_cnt` to obtain CPU fraction consumed by the each thread. We sum all division values and compare it to be less then one, which means total CPU utilization must not exceed 100%. This is source code excerpt from `edf_prep()` which was explained in the previous sub-section:

```
sum+=(100*thread->edf_info->wcet)/thread->edf_info->period
```

We keep doing the same computation inside a while loop for each application threads. If `sum`'s value exceeds 100, it indicates CPU utilization to be higher than hundred percent. This obviously results to non-feasible schedule and causes application rejection. An exact source code excerpt is illustrated in the Figure <4-9>. As it can be seen in the lines from 673 to 677, if `sum` exceeds 100% threshold, kernel will stop execution prompting user with appropriate message "*Non-feasible schedule, pls correct thread's wcet/period.*" However, the most critical part in this approach remains accurateness of the `wcet` value [7, 8]. We presume complexity of having pre-determined `wcet` for the particular thread. Especially, it is very hard and almost impossible in dynamic preemptive environment. We are addressing this issue to be solved by experimental approach and we keep it as our future work.

Chapter Ⅴ. Experimental results

In this chapter we will illustrate experimental results for EDF and NBB kernel. Both of them are implemented in eCos kernel and tested with various scenarios of producer-consumer applications, while MpSc (Multiple Producers and Single Consumer) being OFP prototype application.

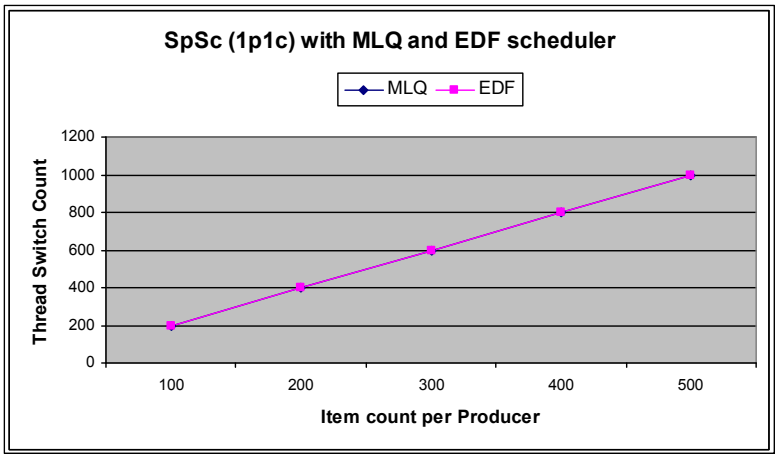
5.1 eCos-EDF performance

EDF scheduler is expected to be more efficient, thus resulting to better kernel characteristics in several points. In our case it is mainly two points, they are re-scheduling needed and preemption introduced by an application. We tested our eCos-EDF kernel for the combinations of Producer-Consumer application, which appears in most of the mainstream computer science textbooks. Below we will illustrate experimental results.

The reason why we compared EDF with MLQ scheduler, is their similarity. As we stated in eCos schedulers explanation section, currently there are only two scheduler policies available in eCos kernel, Bitmap and MLQ. Since Bitmap is static priority thread with one thread per priority, it would not be correct to compare EDF with Bitmap. Also, in Bitmap it is very simple and straightforward to find the highest priority thread, thus we will execute our logic with less overhead. On the other hand, MLQ keeps several threads in the same priority and makes scheduling decision based on ready thread queue, which is similar to EDF, thus more acceptable to compare.

Basically, we have a full right to make an assumption to have similar results with [3]. There authors made comparison between EDF and RM scheduler. RM falls under the static priority scheduling group, where

threads are assigned fixed priority based on their execution frequency [3]. MLQ too belongs to the same group; the difference is only that, in the MLQ scheduler application programmer is supposed to assign unique thread priorities based on his/her preexisting knowledge about application behavior. When we tested eCos with the EDF and MLQ scheduler and had similar indicators when number of threads is less and CPU utilization is low as it is illustrated in Figure <5-1>.

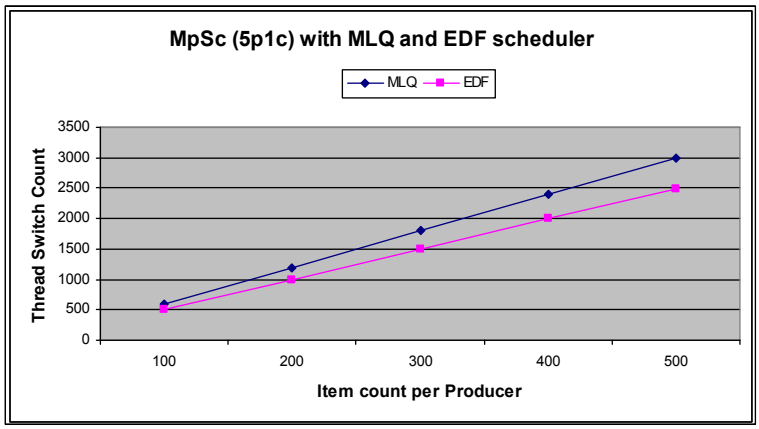


<Figure 5-1> MLQ and EDF in SpSc

Figure above illustrates EDF and MLQ kernel performance with Single producer and Single consumer (SpSc) application. For all tests `Loop Count` on axis `X` is the number of items produced by each producer thread. Axis `Y` represents number of thread switches introduced by an application. Here, overlapped lines means `SpSc` application had the same performance in both kernels. The reason for the same performance is in the number of threads in application. Only two threads will not make big differences for the MLQ and EDF scheduler, thus having the same characteristics.

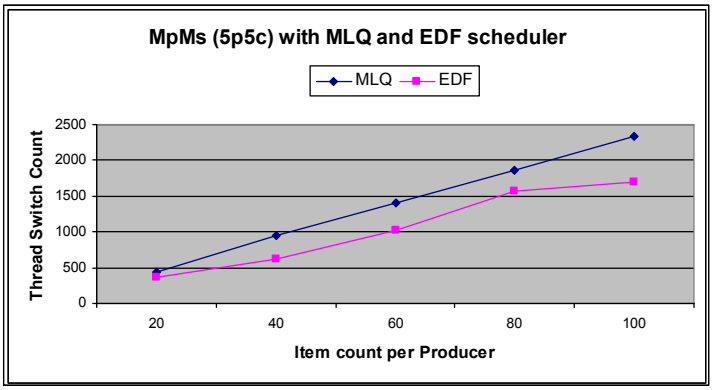
As we increase the number of threads, thus having higher CPU utilization, we start to notice the difference between two schedulers, Figure <5-2>. In this test we have five producers and one consumer thread. As we can see from the figure below, when `Loop Count` is 500,

which means each producer produced by 500 items, resulting to 2500 items for five threads, we have around 3000 context switches in MLQ, while in EDF it is about 2500, resulting to more than 15% performance improvements for this point.



<Figure 5-2> MLQ and EDF in MpSc

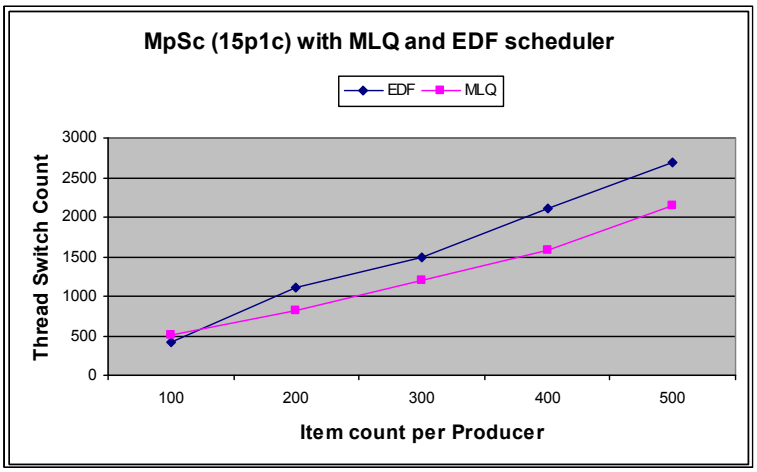
As we continue to increase the number producer and consumer threads, we will have far different picture in two kernel schedulers. Figure <5-3> illustrates five producer and consumer threads' performance. Here we have almost 30% improved kernel performance in terms of thread context switch when loop count is equal to 100.



<Figure 5-3> MLQ and EDF in MpMc

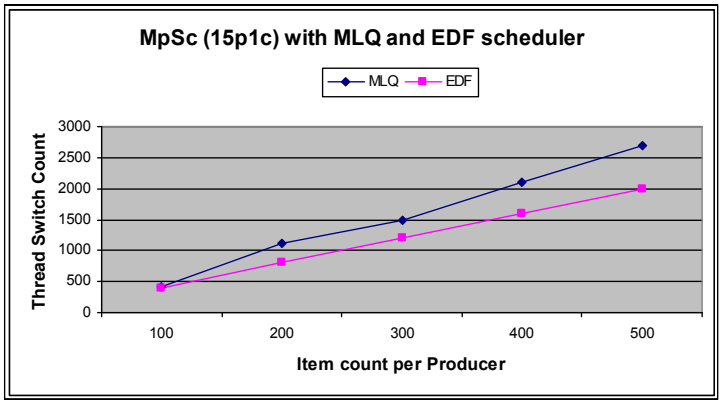
Another point where EDF performs better is the number of preemption in kernel. It is indirect positive shape of less context switch. Preemption occurs when ready higher priority thread preempts executing

lower one and takes CPU ownership, resulting for the lower priority thread to sleep. So, more preemption means the more thread sleep and wake calls, and vice versa. As above figures illustrated less context switches, we will have proportional decline in amount of thread sleep. It mean most of our threads are completing their execution in a one run, without sleep, thus offering a solid deadline keeping feature. As our OFP prototype application is MpSc we made several additional experiments on that. Two of them are illustrated in Figure <5-4> and Figure <5-5>.



<Figure 5-4> MLQ and EDF in MpSc

Figure above the case when number of producer threads increased to 15. Interesting point is regardless number of producer threads (Figure <5-2> and Figure <5-4>) as we produce more than 500 items per producer thread, Thread Switch Count will remain to be decreased about 25% in EDF than it is in MLQ.



<Figure 5-5> MLQ and EDF in MpSc

Figure <5-5> illsutrates MpSc application with 15 producers and 1 consumer with higher frequency than previous figure. In last figure period of producer threads were decreased to 100 msec, while in the figure <5-4> it was 250 msec. Thus, even in high frequency with tough competition for CPU, EDF will provide more than 20% less thread context switches in eCos kernel.

5.2 eCos-NBB performance

Generally, message communication solutions that introduce blocking are penalized by locking that introduces priority inversion, deadlock scenarios and bottlenecks. The time that a process can spend blocked while waiting to get access to the critical section, can form substantial part of the algorithm execution time [4]. Once we will be able to provide lock-free thread communication mechanism for our OFP, we will have less scheduler locks, having more room for other the urgent computation to be executed earlier. In this way, we will have indirectly positive impact on timeliness of the OFP and our kernel as a whole.

In order to test our NBB with the application sharing similar characteristics with OFP, we tested it with producer-consumer application. Figure <5-6> illustrates our NBB compared with eCos

Message Box (MBox) in terms of scheduler locks.

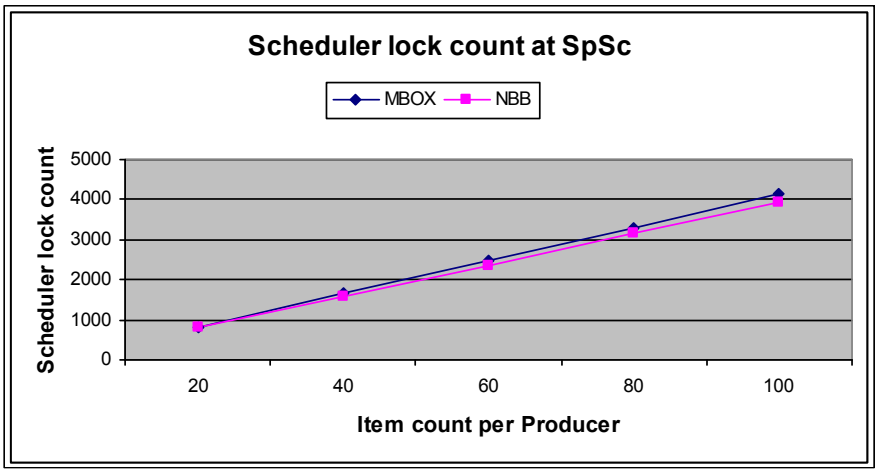


Figure <5-6> NBB VS MBox in SpSc

MBox is one of the message communication mechanisms provided by eCos kernel [22]. In this application, single producer and single consumer (SpSc) threads share the same MBox to exchange messages. As it is indicated in the x axis, totally 100 items were produced and consumed by our producer and consumer threads. Y axis represents number of locks introduced by application at runtime. As we lock the scheduler in the critical section (read or write) of the MBox, we are expected more scheduler locks than it is in NBB (it does not uses locks). As we can see in the Figure <5-6>, scheduler lock count is slightly less than it is in MBox. The reason, why have this amount of locks is, scheduler lock is introduced not only by message communication mechanism, but from all part of the kernel. Scheduler is locked when ASR or ISR is set, thread are made ready, deleted, slept, woken up, suspended, resumed, alarm is created or triggered and etc. As we have all part of the kernel is the same except message communication mechanism, we assure lock difference is introduced by NBB and MBox only. It should be noted that, via single producer and single consumer it is not so illustrative to have different kernel behaviors with two mechanisms. The reason is illustrated in the Figure <5-7>.

In single producer and single consumer, threads have bigger delta time (indicated as `X` in green) between write and read operation of the items, Figure <5-7>. The difference could be more visible, once we will execute NBB as in the Figure <5-8> scenario. To achieve this, we created multiple producers and single consumer (MpSc) to communicate with a shared NBB. In this case dense competition between producers will introduce more scheduler locks, Figure <5-9>.

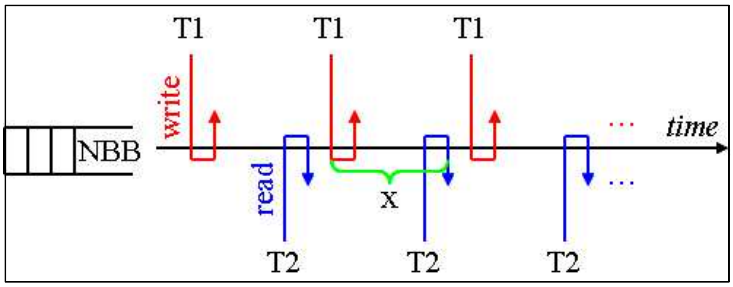


Figure <5-7> NBB with less thread interference

In single producer and single consumer, threads have bigger delta time (indicated as `X` in green) between write and read operation of the items, Figure <5-7>. The difference could be more visible, once we will execute NBB as in the Figure <5-8> scenario.

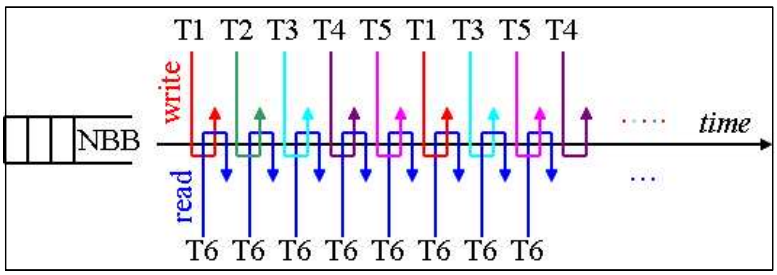


Figure <5-8> NBB with high thread interference

To achieve this, we created multiple producers and single consumer (MpSc) to communicate with a shared NBB. There are five producer threads in the top, indicated with different colors and `T1`, `T2`, `T3`, `T4` and `T5`. All producer threads are running in 10 millisecond frequency (one item is produced in 10 ms) and they have the same priority equal to 4. There is only one consumer thread `T6`, running on 2 ms frequency and

in relatively less priority than producers. Consumer thread has priority value equal to 4. In this scenario, there will be more intereference between consumer and producer threads, thus introducing more scheduler locks via dense competition than in NBB, Figure <5-9>.

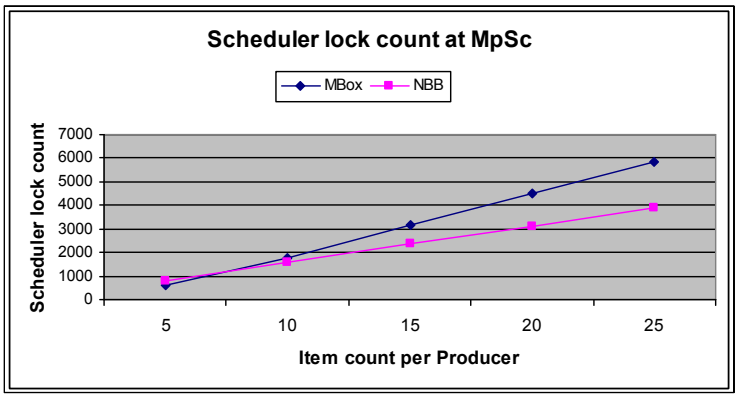


Figure <5-9> NBB VS MBox in MpSc

We created five producer and single consumer thread communicating via single instance of NBB. As we expected, difference in the number of scheduler locks will increase as we continue to increase number of items exchanged via NBB. Figure <5-9> shows number of locks to be almost same, when number of produced items by each thread is equal to 5, resulting to 25 produced items by all five threads. However, once we increase number of produced items to 25 by each thread (resulting to 125 items by all five) we will notice almost 35% less scheduler locks on eCos-NBB. It means our NBB based kernel is 35% responsive than it is in MBox, which will allow eCos-NBB kernel to have finer timeliness to execute timely critical operations. Also, we observed execution time of NBB and MBox based message communication mechanisms remained almost the same (less than 1% difference) during all experimental tests.

As it was mentioned in the second chapter, OFP has shared characteristics with multiple producers and single consumer thread. Based on this, we can conclude similar performance improvements will be achieved at eCos-NBB-OFP.

Chapter Ⅵ. Conclusions

In this work we addressed two issues to enhance RTOS kernel. They are extension of eCos kernel with efficient EDF scheduler and lock-free message communication mechanism called NBB.

We briefly summarized real-time scheduling, eCos itself and its schedulers and provided definition of the problem. Next, we explained our experimental OFP application characteristics carrying real-time computation for small UAV. Also, need for and suitability of EDF and NBB were explained with given periodic characteristics of our practical application, which prototype application was built based on. Related works were discussed with their similarities and difference in implementations and targeted domain. We illustrated fundamental data structure we have used to build EDF scheduling and explained important techniques used during implementation. Implementation details were accompanied with a source code skeletons where it is inevitable. Next, we presented our simple mechanism to make scheduling feasibility test. NBB implementation was also illustrated from the scratch, together with development environment, tools and with the source code skeletons.

Both of our solution were implemented in eCos kernel and tested with various scenarios of producer-consumer applications, which shares practical OFP characteristics. Solutions are provided with an appropriate experimental result. Although we don't illustrate direct implementation and experimental results for our real OFP, we built our prototype application sharing most of the characteristics and computational nature to be as it is in real OFP. Thus, implementation and experimental results illustrated for our prototype application will confirm kernel enhancements to be true for our real OFP as well. Future work will be to implement and experiment eCos-EDF and eCos-NBB in our real OFP.

REFERENCES

[1] Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard real-time environment. JACM 20(1), 46-61 (1973)

[2] K. H. (Kane) Kim, Juan A. Colmenares, Kee-Wook Rim: Efficient Adaptations of the Non-Blocking Buffer for Event Message Communication. ISORC2007 (10th IEEE CS Int′ Symp. on Object & Component Oriented Real-Time Distributed Computing), Santorini, Greece, May 7-9, 2007, pp. 29-40.

[3] Giorgio C. Buttazzo. “Rate Monotonic vs. EDF: Judgment Day”. Real-Time Systems, 29(1):5-26, 2005.

[4] P. Tsigas and Y. Zhang. A simple, fast and scalable nonblocking concurrent FIFO queue for shared memory multiprocessor systems. In Proc. of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'01), pp. 134-143, 2001.

[5] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. Journal of Parallel and Distributed Computing, 51(1):1-26, 1998.

[6] Sanjoy Baruah, Shelby Funk, and Joel Goossens: Robustness Results Concerning EDF Scheduling upon Uniform Multiprocessors. IEEE Transactions on Computers, Vol. 52, No. 9, pp. 1185-1195, September 2003

[7] Alan Burns, Andy J. Wellings and Fengxiang Zhang. “Combining EDF and FP Scheduling Analysis and Implementation in Ada 2005”. Lecture Notes In Computer Science; Vol. 5570, Springer-Verlag, ISBN:978-3-642-01923-4, pp. 119-133, 2009.

[8] Jansen, Pierre G. and Mullender, Sape J. and Havinga, Paul J.M. and Scholten, Hans. Lightweight EDF Scheduling with Deadline Inheritance. Internal Report, University of Twente Publications, 2003.

[9] eCos home page - http://ecos.sourceware.org/

[10] Embedded software development with eCos, Anthony J. Massa, Prentice Hall, 2002.

[11] Modern Operating Systems, by Andrew S. Tanenbaum, Pearson Education, 2nd edition, pp. 132-152, pp. 473-475, 2001.

[12] Nodir Kodirov, Doo-Hyun Kim: A Design Strategy for Enhancing eCos with EDF for Disaster Response System. International conference on IT Promotion in Asia 2009 in Conjunction with International Summit on Information and Communication Technologies. pp. 212-216. Tashkent, Uzbekistan, September 21-25, 2009.

[13] Ubuntu homepage - http://www.ubuntu.com/

[14] Scintilla and SciTE - http://www.scintilla.org/

[15] Doo-Hyun Kim, Nodir Kodirov, Chun-Hyon Chang, Jung-Guk Kim. “HELISCOPE Project: Research Goal and Survey on Related Technologies”. ISORC 2009-12th IEEE International Symposium on Object/Component/service-oriented Real-time distributed Computing, pp. 112-118. Tokyo, Japan, March 17-20, 2009.

[16] Robert I. Davis and Alan B. “A Survey of Hard Real-Time Scheduling Algorithms and Schedulability Analysis Techniques for Multiprocessor Systems”, Technical Report available at http://www.cs.york.ac.uk/ftpdir/reports/2009/YCS/443/YCS-2009-443.pdf

[17] Y-H Chao, S-S Lin, K-J Lin, “Schedulability issues for EDZL scheduling on real-time multiprocessor systems”, Information Processing Letters, Volume 107, Issue 5, pp. 158-164, 2008

[18] S.K. Baruah., A. Burns, “Sustainable Scheduling Analysis”. In Proceedings of the IEEE Real-Time Systems Symposium, pp. 159-168, 2006.

[19] T.P. Baker, S.K. Baruah, “Schedulability Analysis of global EDF”, Real-Time Systems, 38: pp. 223-235, 2008.

[20] M. Bertogna, M. Cirinei, G. Lipari, “Improved schedulability analysis of EDF on multiprocessor platforms”. In Proceedings of the 17th Euromicro Conference on Real-Time Systems, pp. 209-218, 2005.

[21] James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay, “Real-Time Computing with Lock-Free Shared Objects”, In Proceeding of the 16th IEEE Real-Time Systems Symposium, Pisa, Italy, 1995, pp. 28-37

[22] eCos Reference Manual - http://ecos.sourceware.org/docs-latest/ref/ecos-ref.html