MÄLARDALEN UNIVERSITY
SWEDEN

Virtual Clustered-based Multiprocessor Scheduling in Linux Kernel

Master Thesis

Author:
Syed Md Jakaria Abdullah
sah11001@student.mdh.se

Supervisor:
Nima Moghaddami Khalilzad
Examiner:
Moris Behnam

School of Innovation, Design and Engineering (IDT)
Mälardalen University
Västerås, Sweden

7th July 2013

Abstract

Recent advancements of multiprocessor architectures have led to increasing use of multiprocessors in real-time embedded systems. The two most popular real-time scheduling approaches in multiprocessors are global and partitioned scheduling. Cluster based multiprocessor scheduling can be seen as a hybrid approach combining benefits of both partitioned and global scheduling. Virtual clustering further enhances it by providing dynamic cluster resource allocation during run-time and applying hierarchical scheduling to ensure temporal isolation between different software components. Over the years, the study of virtual clustered-based multiprocessor scheduling has been limited to theoretical analysis. In this thesis, we implemented a Virtual-Clustered Hierarchical Scheduling Framework (VC-HSF) in Linux without modifying the base Linux kernel. This work includes complete design, implementation and experimentation of this framework in a multiprocessor platform. Our main contributions are twofold: (i) to the best of our knowledge, our work is the first implementation of any virtual-clustered real-time multiprocessor scheduling in an operating system, (ii) our design and implementation gives practical insights about challenges of implementing any virtual-clustered algorithms for real-time scheduling.

Acknowledgements

First of all I would like to thank my thesis supervisor Nima Moghaddami Khalilzad and thesis examiner Moris Behnam for giving me the opportunity to do this interesting thesis. Special thanks to Svenska Institutet (SI) for sponsoring my education in Sweden. Finally, I thank my loving family who always provided mental support while studying in Sweden for past two years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

In recent years, we have witnessed a major paradigm shift in the computing platform design. Instead of increasing the running frequency of processors to improve the performances of processing platforms, the hardware vendors now prefer to increase the number of processors available in a single chip. Single-core chip designs suffer many physical limitations such as excessive energy consumption, chip overheating, memory size and memory access speed. These problems can be reduced by placing multiple processing cores that share some levels of cache memories on the same chip. In the general-purpose arena, this trend is evidenced by the availability of affordable Symmetric Multiprocessor Platforms (SMPs), and the emergence of multicore architectures. In the special-purpose and embedded arena, examples of multiprocessor designs include network processors used for packet-processing tasks in programmable routers, system-on-chip platforms for multimedia processing in set-top boxes and digital TVs, automotive power-train systems, etc. Most of these multiprocessor based embedded systems are inherently real-time systems [10]. A major reason for the proliferation of multicore platforms in real-time systems is that such platforms now constitute a significant share of the cost-efficient Components-Off-the-Shelf (COTS) market. Another important factor is their considerable processing capacity, which makes them an attractive choice for hosting compute-intensive tasks such as high-definition video stream processing. If the current shift towards multicore architectures by the major hardware vendors continues, then in near future, the standard computing platform for real-time embedded system can be expected to be a multiprocessor. Thus multiprocessor-based software designs will be inevitable.

The study of real-time scheduling in multiprocessors dates back to early 70s [42], even before the appearance of the actual hardware. Recent advancement of the processing platform architectures led to a regain of interest for the multiprocessor real-time scheduling theory during the last decade. Even though the real-time scheduling theory for uniprocessor platforms can be considered as being mature, the real-time multiprocessor scheduling theory is still an evolving research field with many problems remained open due to their intrinsic difficulties. A detail survey of real-time multiprocessor scheduling up to 2009 can be found in [22].

Two most popular approaches in multiprocessor scheduling are global scheduling and partitioned scheduling [22]. In partitioned scheduling a task set is partitioned into disjoint partitions and each of the partition is independently scheduled in its assigned processor. In contrast, global scheduling uses only one global scheduler to schedule all the tasks in all the available processors.

However, partitioned scheduling suffers from inherent algorithmic complexity of partitioning and global scheduling is not scalable due to scheduler overhead. Recently, a number of hybrid approaches combining both partitioned and global scheduling methods have been proposed such as semi-partitioned [7] and clustered scheduling [16]. Clustering reduces the partitioning problem by making smaller number of large partitions (clusters of processors) and distributes overhead of global scheduler into per-cluster schedulers. Moreover, clustered scheduling can schedule task sets that can not be scheduled using partitioned and global scheduling. For example, let us consider a sporadic task set comprised of 6 tasks as follows: $\tau_1 = \tau_2 = \tau_3 = \tau_4 = (3,2,3)$, $\tau_5 = (6,4,6)$ and $\tau_6 = (6,3,6)$ where the notion (T,C,D) denotes minimum arrival time (T), worst-case execution time (C) and relative deadline (D) respectively. This task set is not schedulable under any partitioned and global scheduling on a multiprocessor platform with 4 processors. However, the same task set can be scheduled using clustered scheduling as follows: tasks τ_1, τ_2 and τ_3 can execute on a cluster C_1 comprised of 2 processors, and tasks τ_4, τ_5 and τ_6 can execute on another cluster C_2 comprised of 2 processors. Figure 1.1 shows an example schedule of this task set from [50], where other global scheduling algorithms like global Earliest Deadline First (EDF) [43], EDZL [38], Least Laxity First (LLF) [46], fp-EDF [12] and US-EDF[m/2m-1] [52] failed but clustered scheduling can meet all the task deadlines.

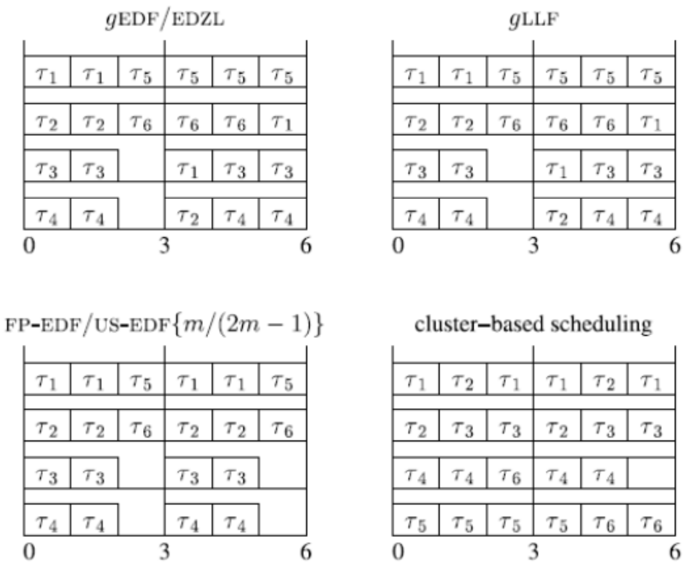

Figure 1.1: Example of clustered scheduling [50]

The notion of physical cluster of processors is enhanced by Shin et al. [50] using virtual clusters. Virtual clusters are dynamically mapped into a set of available processors and it uses hierarchical scheduling. This dynamic mapping allows virtual clusters to utilize processor time more efficiently during run-time. Additionally, more tasks can be added to a virtual cluster easily if the cluster has slack processor time in any of its processors. Therefore, virtual clustered scheduling appears to be more flexible than the original physical clustering as used by Calandrino et al. [16]. Although Easwaran et al. [25] provided a complete hierarchical scheduling framework for implementing virtual clustering, there is no experimental implementation of it to the best of our knowledge.

1.2 Related Works

1.2.1 Clustered Multiprocessor Scheduling

The $LITMUS^{RT}$(Linux Testbed for Multiprocessor Scheduling in Real-Time systems) [17] project is a patch based extension of the Linux kernel with support to test different multiprocessor real-time scheduling and synchronization protocols. This experimental platform supports the sporadic task model and different scheduling mechanisms can be implemented as modular plug-ins.

In $LITMUS^{RT}$, C-EDF (Clustered EDF) algorithm is implemented to compare its performance with respect to other multiprocessor algorithms [14]. The main idea of C-EDF is to group multiple processors that share a cache (either L2 or L3) into clusters and assign tasks to each of them offline. Each cluster has a separate runqueue and during runtime uses a global scheduling algorithm within the cluster. Tasks can only migrate between the processors of their cluster and different clusters do not share processors. Indeed, this implementation of C-EDF is an example of physical clustered scheduling in multiprocessor.

Lelli et al. [39] implemented clustered scheduling in a multiprocessor extension of the customized Linux scheduling class `SCHED_DEADLINE` [26]. Their implementation of C-EDF relies on patch based modification of the kernel via a new scheduling class which uses default migration techniques offered by the Linux kernel. The major difference between [17] and [39] is that the later implementation conforms more with the `POSIX` standard of Linux and thus requires less modification to the original kernel.

However, the focus of our work is virtual clustered scheduling which differs from C-EDF in several aspects. Firstly, unlike physical clusters, virtual clusters can share processors. Secondly, instead of assigning processors to a cluster offline, virtual clusters can be assigned online using global scheduling. Finally, the task migration is not limited to a set of processors (like clustered processors of C-EDF) as processors assigned to a cluster can change dynamically.

1.2.2 Hierarchical Multiprocessor Scheduling

Two-level hierarchical scheduling [23] that has been introduced for uniprocessor platforms provides a temporal isolation mechanism for different components (subsystems). This is increasingly becoming important due to the component-based nature of systems' software development. There are several models to abstract the resource requirements of software components such as the bounded delay model [47] and the periodic resource model [51]. These resource models are extended to hierarchical multiprocessor scheduling such as the Multiprocessor Periodic Resource (MPR) model [50] and the Bounded-Delay Multipartition (BDM) [41] model. Both the MPR and BDM models are proposed as part of a hierarchical scheduling framework which comprises schedulabilty analysis, resource interface generation and run-time allocation. As shown in the BDM, in the original MPR it is assumed that the servers on different processors are synchronized. However, this assumption is relaxed in [35]. To the best of our knowledge none of these frameworks is actually implemented in a multiprocessor platform.

Checconi et al. [19] has an implementation of Two-level hierarchical scheduling for multiprocessors in Linux. Their implementation exploits hierarchical resource management and task group scheduling support in Linux via `cgroups` [1] and `throttling` [3] mechanisms. However, their implementation of hierarchical scheduling requires multiple global schedulers and each of the subsystem has access to all the processors. Additionally, this implementation is patch based thus requires modification of base Linux kernel.

Different implementation schemes for hierarchical scheduling in multiprocessor are analysed in [8]. Later authors of this work [8] have implemented hierarchical scheduling for uniprocessor using a modular scheduler framework called ExSched [9]. ExSched requires no modification of underlying operating system and supports plug-in based development of different schedulers. Several scheduler plug-ins supporting different scheduling schemes has been implemented for both Linux and VxWorks. The main idea of ExSched is to use a loadable kernel-space module to provide different real-time scheduling schemes on top of native Linux scheduling classes. As this approach of implementation is highly configurable compared to other patch-based approaches [19] [17], we will use it in our implementation. A detailed description of how ExSched works can be found in the Section 3.2.

Hierarchical compositional scheduling has been realized in [53] and [37] through virtualization. However, our work is different from these papers in two aspects. Firstly, we intend to implement the complete hierarchy of schedulers within a single operating system. Secondly, none of the previous works [53], [37] addressed the MPR interface as the resource interface model.

1.3 Thesis Objective

1.3.1 Aim of this Thesis

- To analyse virtual-clustered scheduling from implementation point of view and find different design challenges related to it.
- To implement a virtual-clustered scheduler in the Linux kernel with minimal modification to the original scheduler.
- To measure overhead of a virtual-clustered scheduler and determine scopes of optimization.

1.3.2 State of the Art Challenges

- To the best of our knowledge this work is the first attempt to implement virtual-clustered hierarchical scheduling in an operating system. There is no other prior implementation of real-time multiprocessor scheduling that completely resembles our design challenges.
- To the best of our knowledge, there is only one other implementation of hierarchical multiprocessor scheduling [19]. So this thesis is addressing a state of the art research topic.
- Our intended implementation of the Virtual-Clustered Hierarchical Scheduling Framework (VC-HSF) extends general uniprocessor hierarchical scheduling in two ways; firstly, VC-HSF is for multiprocessor platforms, secondly it has extra level of hierarchy called cluster. Given a set of periodic tasks $\Gamma(\Gamma = \tau_i^j | \forall i = 1,..,n)$ where n is the number of tasks and j is the cluster which it belongs to, a system designer can provide the cluster configuration

according to the MPR model. Our VC-HSF should provide both inter-cluster scheduling and intra-cluster scheduling. An overview of our proposed implementation of VC-HSF is presented in the Figure 1.2. Figure 1.2 shows how two virtual clusters each having 3 tasks can be scheduled in 4 CPUs. Each of the cluster has 2 servers, an inter-cluster scheduler schedules these servers to CPUs. Each cluster has its own intra-cluster scheduler which determines the task that run using the cluster budget.

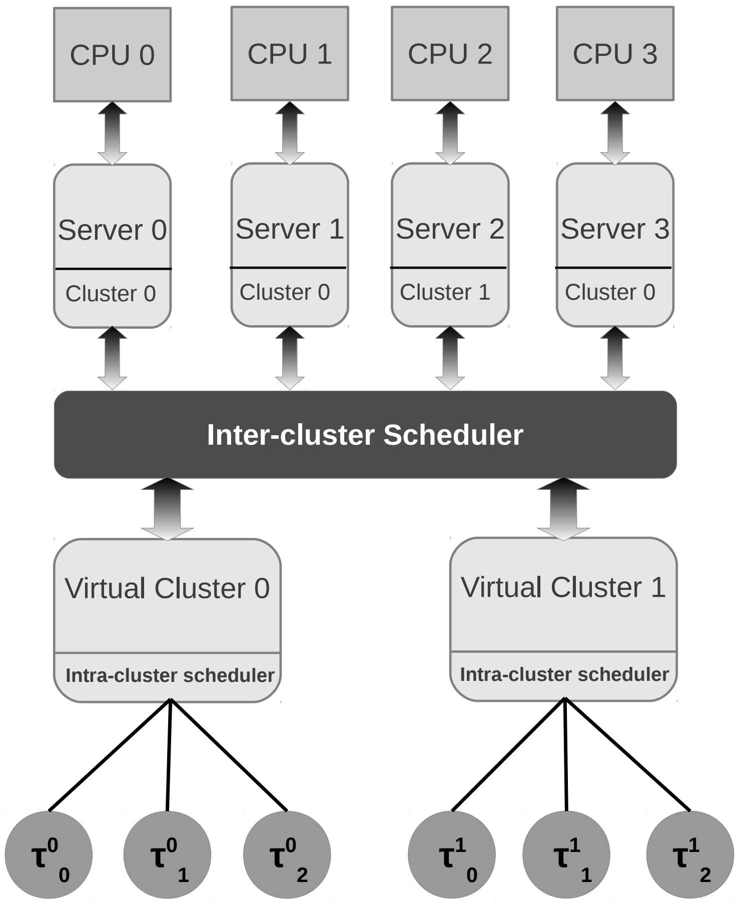

Figure 1.2: Overview of VC-HSF

1.4 Outline of the report

The rest of the report is organized as follows. **Chapter 2** presents brief overview of the state of the art research in real-time multiprocessor scheduling. **Chapter 3** presents background on scheduling mechanism of Linux kernel and ExSched scheduler framework. **Chapter 4** presents detailed analysis of design issues and choices used in our implementation. **Chapter 5** presents experimental evaluations done using our implementation. **Chapter 6** concludes this thesis with a summary and future possible extension of the work presented. **Appendix** provides basic guideline for running VC-HSF and the source code of the main source file of our implementation.

Chapter 2

State of the Art

In this chapter, we present the state of the art in the real-time multiprocessor scheduling research. First we briefly introduce major important terms related to real-time scheduling that is used throughout this chapter.

2.1 Terminology

Real-time Task: A real-time task is a task whose successful execution depends on meeting its timing constraints. Timing constraints of a real-time task are represented by the tuple (T, C, D) with following parameters:

- **Period (T):** The time interval during which a single instance of the task needs to be executed. Each such instance of the task is termed as a **job** of this task.
- **Worst case execution time (C):** The maximum possible execution time needed by the task during a single period.
- **Relative deadline (D):** The time interval relative to the start time of the period during which the task must finish its execution. A real-time task with T = D is called an **implicit-deadline** task.

There are several other important properties of a real-time task such as:

- **Priority:** The priority of a task that determines which task will execute in a time instant.
- **Release time:** The instant in time when a task job becomes ready for execution, typically during start of the period.
- **Absolute deadline:** The time instant relative to release time of a job before which the job must be finished. The real-time task that always needs to meet its absolute deadline is called **Hard** real-time task. On the other hand, the real-time task which can occasionally miss its absolute deadline is called **Soft** real-time task.
- **Utilization:** The utilization of a task is defined as the ratio of the worst case execution time and the period of that task.

There are three different type of real-time tasks:

- **Periodic task:** The task which is released during each start of its period.
- **Sporadic task:** The task which has no period but has a minimum time interval between two successive job releases.
- **Aperiodic task:** The task which can be released any time, thus has no period or minimum inter-arrival time between jobs.

Real-time task scheduling: A real-time task scheduling algorithm is said to be work-conserving if it does not permit there to be any time at which a processor is idle and there is a task ready to execute. There are several other important classifications of scheduling algorithms as:

- **Preemptive or nonpreemptive:** In preemptive scehduling tasks can interfere each others execution. In contrast, in nonpreemptive scheduling when a task starts executing no other tasks can interfere it.
- **Priority based scheduling:** In priority based scheduling, the ready task with highest priority executes first. Three types of priority based scheduling is available [18]:
  1. Fixed task priority: Each task has a fixed priority applied to all its jobs. For example, in the Rate Monotonic (RM) scheduling, priority of task is the inverse of its period.
  2. Fixed job priority: Different jobs of a task can have different priorities but each job has a fixed priority. For example, in the Earliest Deadline First (EDF) scheduling, priority of a task job is fixed by its absolute deadline.
  3. Dynamic job priority: Job of a task can have different priorities depending on its execution period. For example, in the Least Laxity First (LLF) scheduling, priority of a job depends on its laxity or remaining execution time in the execution period.

2.2 Multiprocessor Scheduling

In this section we give a brief overview of state of the art research in real-time multiprocessor scheduling. First we present the state of the art research work in two most popular real-time multiprocessor scheduling algorithms called partitioned scheduling and global scheduling. Then we present the state of the art in hybrid multiprocessor scheduling approaches which combines concepts from both partitioned and global scheduling. These hybrid approaches are semi-partitioned and cluster based scheduling. The section ends with an overview of virtual clustered scheduling which is the focus of our thesis.

2.2.1 Partitioned Scheduling

In partitioned scheduling, a task set is divided into multiple disjoint sets and each of these sets is assigned to a dedicated processor. Processors have their own scheduler with a separate run queue and no migration of task or job is allowed during run time. From a practical perspective, the main advantage of using a partitioning approach to multiprocessor scheduling is that, once an allocation of tasks to processors has been achieved, existing real-time scheduling techniques and analyses for uniprocessor systems can be applied. Advantage of partitioned scheduling in the context of multiprocessor systems first introduced by Dhall and Liu [24] where they showed that there exist task sets with total utilizations arbitrarily close to 1 that are not schedulable by global scheduling even if there are more than one processor in the platform.

However, the reuse of existing results of uniprocessor scheduling theory comes at a price. To obtain m simpler uniprocessor scheduling problems from a multiprocessor platform consisting of m processors, the task set must first be partitioned, that is, each task must be statically assigned to one of the partitions such that no processor is overloaded. Solving this task assignment problem is analogous to the bin packing problem which is known to be NP-Hard in the strong sense [30]. The bin-packing decision problem can be reduced to task-set partitioning in polynomial time in the sense that an implicit-deadline task set is feasible on m processors under partitioned scheduling if and only if there exists a *packing* of all tasks into m bins or processors. Furthermore, the partitioned scheduling algorithms are limited by the performances of the partitioning algorithms used to partition the tasks between the processors of the platform. Indeed, a bin packing or partitioning algorithm cannot guarantee to successfully partition a task set with a total utilization greater than $(m+1)/2$ on a platform composed of m processors [6]. Hence, in the worst-case, a partitioned scheduling algorithm can use only slightly more than 50% of the processing capacity of the platform to actually execute the tasks. For example, an implicit-deadline task system τ with utilization $U_{sum}(\tau)$ could require up to $\lceil 2U_{sum}(\tau) - 1 \rceil$ processors in order to be schedulable using partitioned EDF [44]. In other words, up to half of the total available processor time can be unused under partitioned EDF in the long run. As a consequence, partitioned schedulers may require more processors to schedule a task system when compared to global schedulers. This is clear from the fact that partitioned scheduling algorithms are not work-conserving, as a processor may become idle, but cannot be used by ready tasks allocated to a different processor.

Early research into partitioned multiprocessor scheduling examined the use of common uniprocessor scheduling algorithms such as EDF or Rate Monotonic (RM) on each processor, combined with bin packing heuristics such as First Fit (FF), Next Fit (NF), Best Fit (BF), and Worst Fit (WF), and task orderings such as Decreasing Utilisation (DU) for task allocation. Later different variants of EDF and fixed priority algorithms are proposed such as EDF-Utilization Separation (EDF-US), EDF-First Fit Increasing Deadline (EDF-FFID), etc to improve utilization bound of the partitioned scheduling. A comprehensive view of all these work can be found in the survey by Davis and Burns [22]. Overall, realtime multiprocessor scheduling is difficult from practical point because of intrinsic complexity to partition the task set.

2.2.2 Global Scheduling

In global scheduling, tasks are scheduled from a single priority queue and may migrate among processors. The main advantage of global scheduling is that it can overcome the algorithmic complexity inherent in partitioned approach. As all the processors use a single shared ready queue, this eliminates the need to solve the task assignment problem, which is the source of complexity under any partitioned scheduling. Another key advantage of global scheduling is that it typically requires fewer preemptions as the scheduler will only preempt a task if there is no idle processor. Global scheduling is more suitable for open systems where new tasks arrive dynamically, as a new task can be added easily to existing schedule without assigning it to a particular partition.

However, unlike partitioned scheduling, results from uniprocessor scheduling does not fit easily for global scheduling of multiprocessors. The problem of global scheduling of real-time tasks in multiprocessors was first considered by Dhall and Liu [24] in the context of the periodic task model. Their result known as Dhall's effect shows that neither RM nor EDF retains its respective optimality property in uniprocessor when the number of processors m exceeds one. Given these early negative results and the lack of widespread availability of shared-memory multiprocessor platforms, interest in the global scheduling was quite limited in the first two decades of research into real-time systems [22].

Recently, Phillips et al. [48] showed that the Dhall's effect is more of a problem with high utilization tasks than it is with global scheduling algorithms. This result renewed the interest in global scheduling algorithms. Hence, this property was exploited by $EDF-US[\zeta]$ [52] and $EDF^{(k)}$ [31] to overcome the restrictions of global EDF (gEDF) caused by high utilization tasks. The scheduling algorithm $EDF-US[\zeta]$ always gives the highest priority to the jobs released by tasks with utilizations greater than a threshold ζ. On the other hand, $EDF^{(k)}$ provides the highest priority to the $(k-1)$ tasks with the highest utilizations. In both cases, all other tasks are normally scheduled with gEDF. Later it was proven by Baker [11] that both of these global scheduling algorithms have a utilization bound of $(m+1)/2$ when $\zeta = 1/2$ and k is fixed to an optimal value denoted by k_{min}. This result implies that the two aforementioned global variations of EDF have the same utilization bound like partitioned EDF. Therefore, the first designed global and partitioned extensions of EDF were not able to utilize more than 50% of the platform capacity in the worst-case scenarios. However, partitioned EDF and the various variations of global EDF are incomparable as there exist task sets that are schedulable with the partitioned version but not the global scheduling extension and vice versa. There are many other variants of global scheduling algorithms such as FPZL, FPCL, FPSL, EDZL, EDCL, etc, which mainly improves the schedulability of task sets. In contrast to the partitioned scheduling, some global schedulers incur no utilization loss in implicit-deadline systems. As a result, there exist optimal global schedulers for implicit-deadline tasks, with regard to both hard and soft real-time constraints.

The Proportionate Fair (Pfair) algorithm for implicit deadline periodic task set was introduced by Baruah et al. [13]. Pfair is based on the idea of fluid scheduling, where each task makes progress proportionate to its utilization. Pfair scheduling divides the timeline into equal length quanta or slots. At each time quanta, the schedule allocates tasks to processors, such that the accumulated processor time allocated to each task optimize the overall utilization. However, a Pfair scheduler incurs very high overheads by making scheduling decisions at each time quanta. Further, all processors need to synchronize on the boundary between quanta when scheduling decisions are taken which is hard in practice. Practical implementation of Pfair scheduling [32] on a symmetric multiprocessor showed that the synchronized rescheduling of all processors every time quanta caused significant bus contention due to data being reloaded into cache. To reduce this problem Holman and Anderson [32] proposed the staggered quanta approach where instead of synchronizing at every quanta, tasks required synchronization at smaller number of quantas. However, this approach reduces the schedulability of task sets under this Pfair algorithm and scheduler overhead is still significant.

Using the concept of Pfair algorithm, different variants of proportionally fair algorithms are proposed. Original Pfair algorithm is not work conserving, ERFair [5] removes this problem by allowing quanta of a job to execute before their PFair scheduling windows provided that the previous quanta of the same job has completed execution. PD [13], and PD^2 [4]] improved on the efficiency of Pfair by partitioning tasks into different groups based on utilization. Zhu et al. [54] further reduced scheduling points of Pfair in the Boundary Fair (BF) algorithm by making

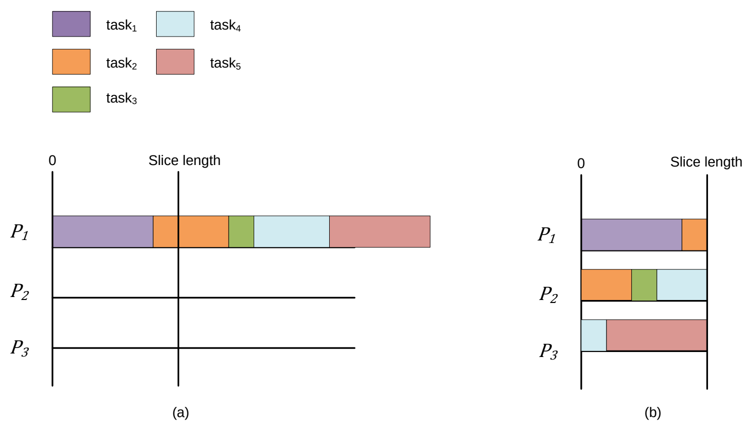

Figure 2.1: McNaughton's algorithm

scheduling decision only at period boundaries or deadlines. This approach is valid for implicit deadline task set, but reduces the fairness property of original Pfair algorithm. Boundary fairness introduced the concept of slices of time for which scheduler can take scheduling decision.

All of these Pfair algorithms are built on a discrete-time model. Hence, a task is never executed for less than one system time unit (which is based on a system tick of operating system). Indeed, many real-time operating systems take their scheduling decisions relying on this system tick. It is therefore quite unrealistic to schedule the execution of a task for less than one system time unit. However, imposing to schedule tasks only for integer multiples of the system time unit highly constrains and complicates the optimal scheduling decisions of the scheduling algorithm. Consequently, researchers focussed on the study of continuous-time systems instead of their discrete-time equivalents. In a continuous-time environment, task executions do not have to be synchronized on the system tick and tasks can therefore be executed for any amount of time. This constraint relaxation drastically eases the design of optimal scheduling algorithms for multiprocessor platforms by increasing the flexibility on the scheduling decision points.

The continuous-time counter-part for the Boundary Fair algorithm is named the Deadline Partitioning Fair (DP-Fair) algorithm. The main idea of DP-Fairness is that, it divides the continuous time into slices based on deadlines (thus called deadline partitioned), during each time slice each task is executed for a local execution time which is the product of the task's utilization and the length of the time slice. Then different heuristics can be used to assign tasks to processors so as to ensure that all deadlines will be met. DP-Fairness property formalized by Levin et al. [40] is used in many algorithms such as DP-Wrap [29], NVNLF [27], LRE-TL [28], LLREF [20], etc. For example, DP-Wrap for periodic tasks uses a slight variation of the next fit heuristic which was inspired by McNaughton's wrap-around algorithm [45]. McNaughton's algorithm packs tasks into a time slice on processors one by one. For example five tasks can be scheduled off-line in three processors as seen in Figure 2.1. However, this requires splitting and migration of tasks. A task set of n periodic tasks on m processors can thus experience at most $m-1$ preemptions on slice boundary and n preemptions inside the time slice. On the other hand, there can be at most $2(m-1)$ migrations of tasks across processors as migration on both ways (leaving or arriving) can happen on all processors except the one where a task meets its deadline (or slice boundary). However, if all the tasks are periodic with implicit deadlines, then the pattern of the schedule computed for each time slice is repeated (see Figure 2.2), only the length

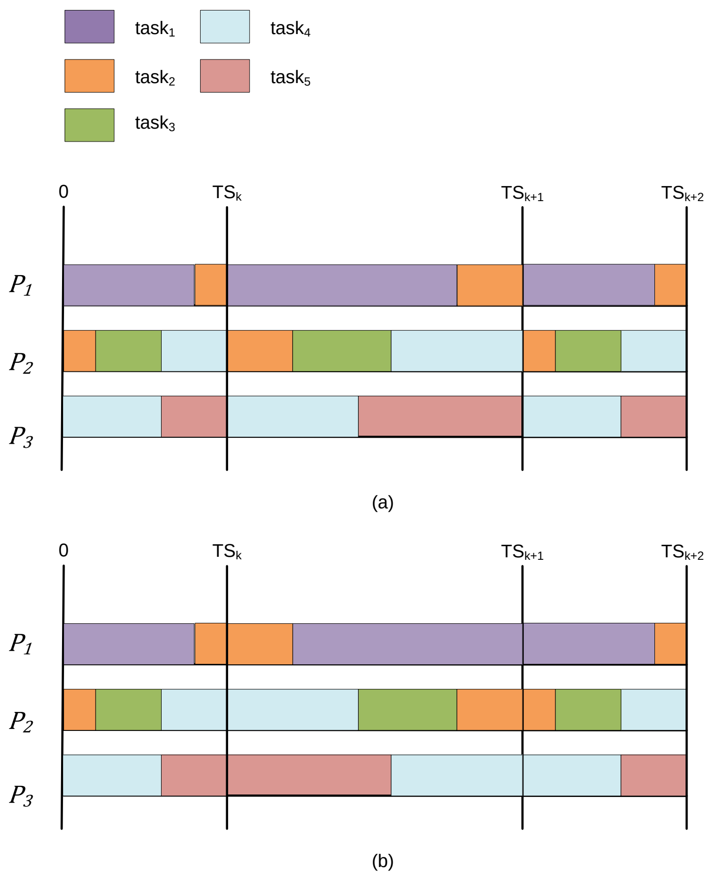

Figure 2.2: DP-Wrap algorithm (a) without mirror (b) mirror mechanism applied

of this schedule may vary. Consequently, the same $m-1$ tasks migrate repeatedly in different slices. Exploiting this property, DP-Wrap reduces the amount of preemptions and migrations via a mirroring mechanism that keeps the tasks that were executing before the boundary TS_k, running on the same processors after TS_k (see Figure 2.2). This technique reduces both the number of preemptions and the migrations by $m-1$, but it is only applicable for the implicit deadline periodic tasks.

There are several other global scheduling algorithms based on laxity or remaining execution time. The Earliest Deadline until Zero Laxity (EDZL) algorithm proposed by Lee [38] is an extension of EDF with an additional feature of raising a task priority to highest when it will miss its deadline unless it executes for all of the remaining time up to its deadline (zero laxity). This idea is used by Kato and Yamasaki [33] in the Earliest Deadline until Critical Laxity (EDCL) algorithm, which increases the job priority on the basis of critical laxity at the release or completion time of a job. Cho et al. [20] presented an optimal algorithm for implicit deadline periodic task sets called the Largest Local Remaining Execution Time First (LLREF). LLREF divides the timeline into sections divided by scheduling events such as task releases or deadlines. A LLREF scheduler selects m tasks with largest local remaining execution time for execution on m processors at the beginning of each such section.The local remaining execution time for a task at the start of a section is the amount of execution time that the task would be allocated during that section in a T-L (time vs laxity) schedule, which is product of length of the section and utilization of that task. The local remaining execution time decrements as a task executes during the section. Funaoka et al. [27] extended the LLREF to work conserving algorithm by using the unused processing time per section. However, LLREF requires large overhead due to large number of scheduling points, task migration and accounting of task laxities. Funk and Nadadur [28] extended the LLREF approach, forming the LRE-TL algorithm by scheduling any task with laxity within a section. This approach greatly reduces the overhead due to migration and LRE-TL is also extended for sporadic task set with implicit deadlines.

Despite considerable research in global multiprocessor scheduling, all the above mentioned algorithms suffers from common disadvantages. As the number of tasks grows in the system, it becomes hard to manage global data structures for task queues. Most of these scheduling algorithms are subject to what is called scheduling anomalies. That is, task sets that are schedulable by a given algorithm S on a platform P become unschedulable by S on P when they are modified such as the reduction of the utilization of a task by increasing its period or reducing its worst-case execution time, finishing the execution of a job earlier than initially expected, adding processors to the platform. Period anomalies are known to exist for global fixed-task priority scheduling of synchronous periodic task sets, and for global optimal scheduling (full migration, dynamic priorities) of synchronous periodic task sets.

2.2.3 Semi-partitioned Scheduling

In global scheduling, the overhead of migrating tasks can be very high depending on the architecture of the multiprocessor platform. In fact delays related to cache miss and communication loads can potentially increase the worst case execution time of a task which is undesirable in real-time domain. On the other hand, fully partitioned algorithms suffer from waste of resource capacity as fragmented resource in each partition remains unused. To overcome this problem, hybrid approaches are proposed which includes semi-partitioned and clustering algorithm.

In semi-partitioned scheduling algorithm, most of the tasks are executed on only one processor as in original partitioned approach. However, a few tasks (or jobs) are allowed to migrate between two or more processors. The main idea of this technique is to improve the utilization bound of partitioned scheduling by globally scheduling the tasks that cannot be assigned to only one processor due to the limitations of the bin-packing heuristics. The tasks that cannot be completely assigned to one processor will be split up and allocated to different processors. The process of assigning tasks to processors is done off-line .

EKG (EDF with task splitting and k processors in a Group) is an optimal semi-partitioned scheduling algorithm for periodic task set with implicit deadlines [7]. It is built upon a continuous-time model and consists in a semi-partitioned approach which adheres to the Deadline Partitioning Fair (DP-Fair) theory. In EKG, the tasks which needed splitting is termed as the migratory task and the other tasks which can be completely assigned to a processor are termed as component tasks. It splits the multiprocessor platform with m processors into several clusters based on parameter k ($k \leq m$). $\lceil \frac{m}{k} \rceil$ clusters are assigned k processors each and a single cluster is given $m - \lceil \frac{m}{c} \rceil k$ processors. A bin-packing algorithm (such as Next Fit) is then used to partition the tasks among the clusters so that the total utilization on each cluster is not greater than its number of constituting processors. After the partitioning of the task set among the clusters, EKG works in two different phases, firstly, the tasks of each cluster are assigned to the processors of its cluster. Then, the tasks are scheduled in accordance with this assignment, using a hierarchical scheduling algorithm based on both the DP-Fairness and EDF. However, it was shown in recent studies that the division of the time in slices bounded by two successive deadlines and the systematic execution of migratory tasks in each time slice inherent in DP-Fair algorithms, significantly reduce the practicality of EKG.

An alternative approach developed by Bletsas and Andersson [15] first allocates tasks to physical processors (heavy tasks first) until a task is encountered that cannot be assigned. Then the workload assigned to each processor is restricted to periodic reserves and the spare time slots between these reserves organized to form notional processors. Kato and Yamasaki [34] presented another semi-partitioning algorithm called the Earliest Deadline Deferrable Portion (EDDP) based on EDF. During the partitioning phase, EDDP places each heavy task with utilization greater than 65% on its own processor. The light tasks are then allocated to the remaining processors, with at most $m-1$ tasks split into two portions. The two portions of each split task are prevented from executing simultaneously by deferring the execution of the portion of the task on the lower numbered processor, while the portion on the higher numbered processor executes.

Semi-partitioning approach is also investigated for fixed priority scheduling and sporadic tasks. Lakshmanan et al. [36] developed a semi-partitioning method based on fixed priority scheduling of sporadic task sets with implicit or constrained deadlines. Their method, called the Partitioned Deadline Monotonic Scheduling with Highest Priority Task Split (PDMS HPTS), splits only a single task on each processor: the task with the highest priority. A split task may be chosen again for splitting if it has the highest priority on another processor. PDMS HPTS takes advantage of the fact that, under fixed-priority preemptive scheduling, the response time of the highest-priority task on a processor is the same as its worst-case execution time, leaving the maximum amount of the original task deadline available for the part of the task split on to another processor.

Although, semi-partitioned algorithms increases utilization bound by using spare capacities left by partitioning via global scheduling, it has a inherent disadvantage of off-line task splitting. It is ongoing state of the art research to efficiently split the tasks with maximum efficiency to reduce overhead related to migration and preemptions.

2.2.4 Cluster Based Scheduling

Cluster based scheduling can be seen as a hybrid approach combining benefits of both partitioned and global scheduling. The main idea of the cluster based scheduling is to divide m processors into $\lceil \frac{m}{c} \rceil$ sets of c processors each [16]. Both partitioned and global scheduling can be seen as extreme cases of clustering with $c = 1$ and $c = m$ respectively.

Initially the notion of clustering is thought to be similar to partitioning approach where the task set is assigned to dedicated processors during an off-line partitioning phase. In case of clustering, this becomes assigning tasks to a particular cluster and give each cluster a set of processors. It simplifies the bin packing problem of partitioning mentioned earlier as now tasks have to be distributed into clusters. Different heuristics can be applied to assign tasks to cluster to improve the utilization, reduce overhead due to migration and response time. Each cluster handles small number of tasks on small number of dedicated processors and thus removes problem of long task queue experienced by the global scheduling algorithms. Clustering also gives flexibility in the form of creating clusters for different types of tasks such as low or high utilization tasks. Another flexibility offered by clustering is that it is possible to create clusters with different resource capacity such as cluster with large or small number of processors, having same second level cache, etc. Shin et al. [50] further expanded this flexibility by analysing cluster based multiprocessor scheduling for virtual clustering. In contrast to the normal clustering approach known as physical clustering where processors are dedicated for a cluster, virtual clustering assigns processors to cluster dynamically during runtime. Shin et al. (2008) proposed the Multiprocessor Periodic Resource (MPR) interface to represent virtual cluster and presented hierarchical scheduling analysis and algorithms for them on symmetric multiprocessor platform. Easwaran et al. [25] extended this hierarchical scheduling framework with optimal algorithms for allocating tasks to clusters. We will present more details about virtual clustering in the following section.

2.2.5 Virtual Cluster Scheduling

The Virtual Cluster (VC) Scheduling framework [50] is a generalization of physical clustering with a new feature of sharing processors between different clusters. Unlike physical clusters, where processors are dedicated to a cluster off-line, VC allows allocation of physical processors to the clusters during run-time. This dynamic allocation scheme requires an interface to capture the execution and concurrency requirements within a cluster to use hierarchical scheduling techniques. The interface proposed by Shin et al. [50] which is known as the MPR model is:

Definition 1. *The MPR model $\mu =< \Pi, \theta, m' >$ where $\theta \leq \Pi$ specifies a unit capacity, identical multiprocessor platform with at most m' processors can collectively supply θ units of execution resource in every Π time units. At any time instance at most m' processors are allocated concurrently to μ where θ/Π denotes the bandwidth of model μ [50].*

In VC scheduling framework, for each cluster C_i a MPR interface μ_i is generated using schedulability analysis presented in Shin et al. [50]. Then each of this interface is transformed into a set of implicit deadline periodic servers for inter cluster scheduling. A time-driven periodic server [21] is defined as $PS_i(Q_i, P_i)$, where P_i is the server period, and Q_i is the server budget which represents the number of CPU time units that has to be provided by the server every P_i time units. The periodic servers idle their budget if there is no active task running inside the server.

One example for mapping the cluster interface to periodic servers is the method proposed by Easwaran et al. [25] which works as follows. Given an MPR interface $\mu_j =< \Pi_j, \theta_j, m'_j >$ for cluster C_j, it creates a set of implicit deadline periodic servers $PS^j_1, \ldots, PS^j_{m'_j}$, where,

$$PS^j_1 = PS^j_2 = \ldots = PS^j_{m'_j - 1} = (\Pi_j, \Pi_j) \tag{2.1}$$

$$PS^j_{m'_j} = (\theta_i - (m'_j - 1).\Pi_j, \Pi_j). \tag{2.2}$$

The servers $PS^j_1, \ldots, PS_{m'_{j-1}}$ are full budget servers, while $PS_{m'_j}$ is a partial budget server.

Once all the interfaces are transformed into the periodic servers, VC uses hierarchical scheduling to schedule servers and tasks. There are two levels of scheduling described in VC, namely inter-cluster scheduling and intra-cluster scheduling. Here the inter-cluster scheduler refers to the global scheduler of the hierarchical scheduling while the notion of intra-cluster scheduler is similar to the local scheduler in hierarchical scheduling.

In hierarchical scheduling, the global scheduler schedules the servers representing the subsystem. The same is true for the inter-cluster scheduler of VC except that each cluster can have up to m' active servers. All the servers from all the clusters are queued according to the global scheduling policy. However, tasks are not assigned to any particular server, rather these only belong to a specific cluster. The intra-cluster executes tasks of the cluster by consuming the budgets of its scheduled servers. Unlike regular hierarchical scheduling, the local or intra-cluster scheduler also has to use a multiprocessor global scheduling algorithm as there can be multiple active servers of a cluster. As a result, VC can be described as global scheduling in two level. Easwaran et al. [25] mentioned different global scheduling algorithms like global EDF (gEDF) and McNaughton's algorithm that can be used in VC.

Now, we present an example to illustrate VC scheduling in more detail. Given a implicit deadline sporadic task set $\tau = \{T_1(2,3), T_2(1,2), T_3(1,3), T_4(1,3), T_5(1,3), T_6(1,3), T_7(1,2), T_8(1,2)$ $, T_9(1,2)\}$ of 9 tasks where the task parameters $T_i(C_i, T_i)$ denote execution time C_i and period T_i respectively. τ is mapped into three virtual clusters C_1, C_2, C_3 using MPR interface $\mu_i = < \Pi_i, \theta_i, m'_i >$ for scheduling in a multiprocessor platform as Figure 2.3. The three MPR interfaces $< 6,7,2 >, < 3,4,2 >, < 2,3,2 >$ are calculated using the schedulability analysis presented in Shin et al. [50].

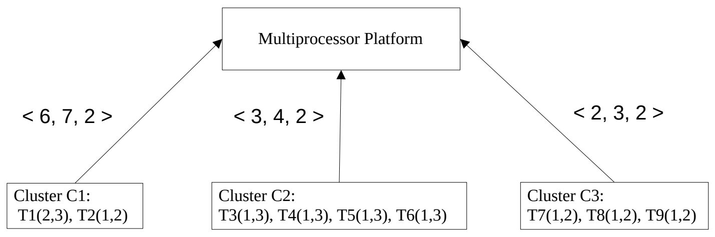
Multiprocessor Platform
< 6, 7, 2 >
< 3, 4, 2 >
< 2, 3, 2 >
Cluster C1: T1(2,3), T2(1,2)
Cluster C2: T3(1,3), T4(1,3), T5(1,3), T6(1,3)
Cluster C3: T7(1,2), T8(1,2), T9(1,2)

Figure 2.3: Virtual cluster mapping using MPR

Virtual clusters C_1, C_2 and C_3 are transformed into periodic servers using 2.1 as shown by the Table 4.1.

Table 2.1: Periodic servers for virtual clusters

	Π	θ	m'	$Servers(PS)$
C_1	6	7	2	$PS_1^1(6,6), PS_2^1(1,6)$
C_2	3	4	2	$PS_1^2(3,3), PS_2^2(1,3)$
C_3	2	3	2	$PS_1^1(2,2), PS_2^1(1,2)$

In Figure 2.4 two examples of inter cluster scheduling of the periodic servers is presented using the McNaughton's algorithm. In Figure 2.4 (a) a simple Next Fit approach is shown by allocating processors to servers sorted by budget requirements and in Figure 2.4 (b) a different schedule is shown for the same set of servers due to problem of server synchronization. It can be seen that a server of a cluster can execute concurrently in at most m processors presented in its MPR interface. Whenever, a server gets activated, it can look for ready tasks (not executing) within its cluster and execute it using the server budget. Though Figure 2.4 (b) shows the advantage of virtual clustering as different clusters can share processors, two disadvantages of this approach is also evident from this example. Firstly, flexibility of sharing processors between clusters comes at the price of overhead related to the synchronization and migration of the servers. For example, in Figure 2.4(b) the second job of PS_2^3 releases before completion of the first job and release of the second job of PS_1^3. If the first job of PS_1^3 completes before the first job of PS_2^2 then PS_2^3 may need to migrate to the available processor with all its tasks. Secondly, similar to the bottleneck of the global scheduling, large utilization servers are less flexible in the schedule and these can effect the schedulability of a task set.

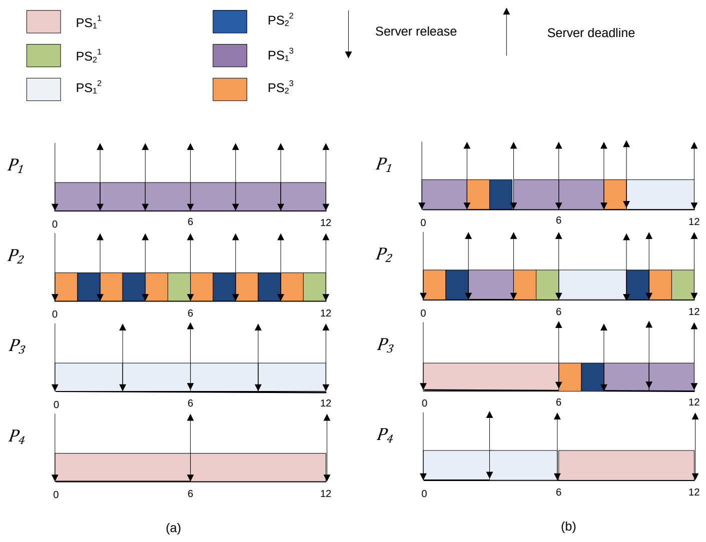

Figure 2.4: Examples of inter cluster scheduling using the McNaughton's Algorithm

However, in our inteneded implementation of VC framework called VC-HSF, we will use global fixed priority algorithm for both server and task scheduling. We first intend to implement VC-HSF for periodic tasks but will allow flexible budget assignment policy to the servers of a cluster.

Chapter 3

Background

In this chapter, we present background on the scheduling in Linux operating system and the ExSched scheduler framework.

3.1 Linux Scheduling Mechanism

In this section we give a brief overview of Linux scheduler and its components [2].

The process scheduler is a component of the Linux kernel that selects which process will run in the processor at any time instant. In a multitasking operating system like Linux, it is the job of the scheduler to distribute processor time to multiple processes in such a way that the user experiences execution of multiple processes simultaneously. However, multiprocessor platforms can run multiple processes in parallel in different processors. Therefore the scheduler has to assign runnable processes to processors and select the running process on each of the processors.

3.1.1 Scheduler Invocation

In the Linux kernel, all scheduling decisions are implemented by the `schedule()` procedure. Its objective is to find a ready process suitable for execution by scheduling policy and then assign the CPU to it. The `schedule()` procedure can be invoked by either directly or in a lazy way by other kernel routines.

In direct invocation, a process invokes the scheduler directly with a call to the `schedule()` procedure when it suspends (e.g., blocked for I/O operation). This call is a blocking call for the calling process and it returns when the process is resumed again by the scheduler.

In lazy invocation, the scheduler is invoked indirectly when preemption of the currently running process is required. This is done by setting the `need_resched` flag in the `task_struct` structure of the running process. The kernel always checks prior to returning from an Interrupt Service Routine (ISR), exception handler or from any system call whether this rescheduling flag is set for the currently scheduled process. If this `need_resched` is set, the kernel invokes `schedule()`. This is useful in a sense that rescheduling can be requested even before calling the scheduler via setting the flag at the end of an ISR or system call. Even in case of non-preemptive execution of a process (which can be achieved in the kernel mode via setting `preempt_count`), rescheduling can be requested which will be served when a non-preemptive execution is over. This gives flexibility to use the ISR even when the running process executes non-preemptively.

3.1.2 Tasks and RunQueue

The Linux scheduler can schedule a task on a processor as a process. It can also schedule a group of processes to achieve hierarchical scheduling (via `cgroup` and `throttling`). All the scheduling entities that a Linux scheduler policy can handle is defined in `sched_entity` structure.

Each process in Linux is represented by a Process Control Block (PCB) called `task_struct`. This structure has fields that distinguish different tasks and their requirements. Some of them are:

- `pid`: a process identifier that uniquely identifies the task.
- `state`: it describes state of the task which can be either unrunnable, runnable or stopped.
- `sched_class *sched_class`: a pointer which binds the task with its scheduling class.
- `cpus_allowed`: a binary mask of the cpus on which the task can run, useful for multiprocessing.

To support both uniprocessors and multiprocessors, the Linux scheduler is organized in a partitioned, per-processor way to ensure cache-local operations. Associated with each processor is a data structure called runqueue (`rq`). It contains a sub-runqueue field for each scheduling class, and every scheduling class can implement its runqueue in a different way. The runqueue contains state of each scheduling class pertaining to that processor (such as a processor-local ready queue, the current time, and scheduling statistics). A ready or runnable process belongs to the runqueue of the processor to which it is currently assigned; when a process suspends, it remains under management of the processor where it was last scheduled until it is resumed.

Each runqueue is protected by a spinlock which must be acquired before its state may be modified (e.g., before processes may be enqueued or dequeued). As there are no per-process locks in Linux to synchronize accesses to `task_struct`s, the runqueue locks are used to serialize process state updates. Each process is assigned to exactly one runqueue at a time, and a processor must first acquire the lock of the assigned runqueue before it may access a `task_struct`. Due to the processor-local nature of runqueues, a scheduler must acquire two (or more) locks whenever a consistent modification of the scheduling state of multiple processes is required. For example, during migration of a process; it must be atomically dequeued from the ready queue of the source processor and enqueued in the ready queue of the target processor. Unless coordinated carefully, such "double locking" could quickly result in deadlock. Therefore, Linux requires that runqueue locks are always acquired in order of increasing memory addresses; that is, once a processor holds the lock of a runqueue at address A_1, it may only attempt to lock a runqueue at address A_2 if $A_1 < A_2$. This imposes a total order on runqueue lock requests; deadlock is hence impossible. Consequently, a processor that needs to acquire a second, lower-address runqueue lock must first release the lock that it already holds and then (re-)acquire both locks. As a result, the state of either runqueue may change in between lock acquisitions.

3.1.3 Linux Modular Scheduling Framework

The Linux scheduler has been designed and implemented by a modular framework that can be easily extended. It is organized as a hierarchy of scheduling classes, where processes of a low priority scheduling class are only considered for execution if all of its higher-priority scheduling classes are idle. At any given time each process belongs to exactly one scheduling class, but can change its scheduling class at runtime via `sched_setscheduler()` system call. Each of these scheduling classes implements specific scheduling policies. Presently there are two high-priority scheduling class for real-time tasks called `SCHED_RR` and `SCHED_FIFO` and three other low-priority scheduling class `SCHED_NORMAL`, `SCHED_BATCH` and `SCHED_IDLE` to provide complete fair scheduling (Linux Kernel 3.2.40).

A Linux scheduling class is implemented through the `sched_class` interface. This interface contains 22 methods which must be implemented for different scheduling classes. While scheduling a process, this methods are called by the scheduler as these are hooked via the `sched_class` structure. Some of the important hook functions are:

- `enqueue_task`: it enqueues a runnable task in the data structure used to keep all runnable tasks called runqueue.
- `dequeue_task`: it removes a task which is no longer runnable from the runqueue.
- `yield_task`: yields the processor giving room to the other tasks to be run.
- `check_preempt_curr`: checks if a task that entered the runnable state should preempt the currently running task.
- `pick_next_task`: chooses the most appropriate task eligible to be run next.
- `put_prev_task`: makes a running task no longer running.
- `select_task_rq`: chooses on which runqueue (CPU) a waking-up task has to be enqueued.

Whenever the Linux scheduler is invoked, it traverses the scheduling class hierarchy in order from the real-time class to the idle class by invoking the `pick_next_task()` method of each class. When a scheduling class returns a non-null `task_struct`, the traversal is aborted and the corresponding process is scheduled. This ensures that real-time processes always take precedence cover non-real-time processes.

In a multiprocessor Linux kernel (configured with `CONFIG_SMP`) has additional fields to support multiprocessor scheduling, such as:

- `select_task_rq`: called from fork, exec and wake-up routines; when a new task enters the system or a task that is waking up the scheduler has to decide which runqueue (CPU) is best suited for it.
- `pre_schedule`: called inside the main schedule routine; performs the scheduling class related jobs to be done before the actual schedule.
- `post_schedule`: like the previous routine, but after the actual schedule.
- `set_cpus_allowed`: changes a given task's CPU affinity; depending on the scheduling class it could be responsible for to begin tasks migration.

A scheduling domain (`sched_domain`) is a set of CPUs which share properties and scheduling policies, and which can be balanced against each other. Scheduling domains are hierarchical, a multi-level system will have multiple levels of domains. A struct pointer struct `sched_domain *sd`, added inside structure `rq`, creates the binding between a runqueue (CPU) and its scheduling domain. Using scheduling domain information the scheduler can make good scheduling and balancing decisions. This is specially useful for partitioned and physically clustered multiprocessor scheduling where a group of processors may share runqueues or scheduling policies.

3.1.4 Priority Based Real-time Scheduling in Linux

Linux support 140 distinct priorities to assign to a process. Of them, the lower 40 ones are reserved for non-realtime tasks and the higher 100 priorities can be used for real-time tasks.

To implement priority based scheduling, each runqueue of a scheduling class is extended with an array of 100 linked lists which is used to queue ready processes at each priority level. To find a high priority process in this list easily, the runqueue contains a bitmap containing one bit per priority level to indicate non-empty lists. This bitmap is scanned for the first non-zero bit using special bitwise operations supported by the hardware; the head of the linked list corresponding to the index of that bit is dequeued. Whenever the dequeue operation results in an empty list, the corresponding bit is reset. This mechanism is commonly called $O(1)$ scheduling of Linux. Although this limits the number of supported priorities to 100, only four instructions is needed in a 32 bit platform to find the highest priority non-empty list. Linux priority based scheduling supports only FIFO and RR approach. In case of FIFO approach, it is implemented by queuing last running task at the end of the linked list corresponding to its priority. The RR algorithm is implemented by doing this enqueue operation when the process finishes its execution quanta.

3.1.5 Task Migration Mechanism

In a multiprocessor platform, Linux uses on demand migration of tasks from one processor to another. This situation arises because the scheduler can schedule a process to any processor permitted by its affinity mask if no other higher priority process is executing on it. In fair based scheduling, this is handled by the load balancing mechanism but real-time schedulers have to consider this in every scheduling decision. A process migration always involves a target and a source processor, either of which may initiate a migration: either a source processor pushes waiting processes to other processors, or a target processor pulls processes from processors backlogged by the higher-priority processes. Processes with processor affinity masks that allow scheduling on only a single processor are exempt from pushing and pulling. However, this pushing or pulling requires taking global spinlocks over two runqueues which is complicated both in terms of synchronization and overhead. The default Linux push and pull mechanisms are not suitable for real-time scheduling because it can cause unnecessary migrations. This is because during the default push and pull operation an idle processor checks all the available processors by processor index in a increasing order. As a result, low priority tasks residing in a processor with lower index can migrate before the actual highest priority one. Due to this undesirable effects, any implementation of the real-time multiprocessor scheduling in Linux should implement its own task migration mechanism on top of the default migration mechanisms.

3.1.6 Hierarchical Scheduling Support in Linux

In hierarchical scheduling [23], tasks are grouped and each group of tasks or subsystem is scheduled individually by its own scheduler. However, each subsystem has an interface comprising of its period, required execution budget and priority which is used by a system wide global scheduler for global scheduling. Global scheduler distributes the available execution time to the servers which represent the lower level subsystems. As a result, hierarchical scheduling has at least two level of schedulers.

Current Linux versions support grouping of tasks via `control group(cgroup)` and `sched-rt-group`. Both of these mechanisms give ability to dedicate a certain share of total execution time to the configured group. For example, `cgroup` associates a set of tasks with a set of parameters for one or more `subsystems`. A `subsystem` is typically a resource controller that schedules a resource or applies per-cgroup limits, but it may be anything that wants to act on a group of processes. Moreover, `hierarchy` is defined as a set of cgroups arranged in a tree, so that every task in the system is exactly in one of the cgroups in the hierarchy, and a set of subsystems; each subsystem has system-specific state attached to each cgroup in the hierarchy. User code may create and destroy cgroups by name in an instance of the cgroup virtual file system, may specify and query to which cgroup a task is assigned, and may list the task PIDs assigned to a cgroup. The intention behind this facility is that different subsystems hook into the generic cgroup support to provide new attributes for cgroups, such as accounting/limiting the resources which processes in a cgroup can access.

The Linux kernel code already provides a rough mechanism, known as CPU `Throttling`, that has been designed for the purpose of limiting the maximum CPU time that may be consumed by individual activities on the system. It was designed so as to prevent real-time tasks to starve the entire system forever. The original mechanism only allowed the overall time consumed by the real-time tasks (irrespective of priority or scheduling policy) to overcome a statically configured threshold, within a time-frame of one second. By default this value (known as `throttling runtime`)is defined to be 950 ms, which is available to all real-time tasks within each second (termed as `throttling period`). By combining the `throttling` and `cgroup` mechanism coarse-grained hierarchical group scheduling can be implemented using the default Linux kernel. In this hierarchical group scheduling, whenever a processor becomes available, the scheduler selects the highest priority task in the system that belongs to any group that has some execution budget available, then the execution time for which each task is scheduled is subtracted from the budget of all the groups that it hierarchically belongs to. The budget limitation is enforced hierarchically, in the sense that, for a task to be scheduled, all the groups containing it, from its parent to the root group, must have some budget left in a non-decreasing order. Together with this group scheduling, `per-group-throttling` mechanism can ensure that none of the groups overrun their assigned execution budget. However, these mechanisms have following drawbacks:

1. The budget and the period assigned initially to a group is the same on all the processors of the system, and is selected by the user. So, in case of multiprocessors, a local server can not be assigned budget only to a particular processor. However, actual execution of the task/process can be limited to a processor via its `cpus_allowed` mask.

2. Default mechanisms of throttling or cgrouping only limits the CPU time consumed by the tasks, it does not enforce its provisioning, nor it has a form of admission control, that is

the total sum of the actual processor utilizations can be greater than 1.

3. The default implementation enforces temporal encapsulation on the basis of a common time granularity for all the tasks in the system, that is one second. This makes time granularities for the tasks quite long, in the order of 1s-10s. As a result, this makes it impossible to guarantee good performance for real-time tasks or servers that need to exhibit sub-second activation and response times.

Although all of the above mentioned limitations can be removed but that effort requires modification of the original Linux kernel. Checconi et al. [19] provided a patch to default Linux throttling mechanism to overcome some of these limitations (except the limitation 1). However, as our goal is to not to modify the default Linux kernel, we are not using any patches.

In summary, we need patch based modification of Linux kernel to implement hierarchical real-time scheduling using cgroups. In this work, we are not using cgroups due to this practical reason.

3.2 ExSched Scheduler Framework

ExSched [9] is a scheduler framework that can be used to implement custom schedulers as plug-ins for different operating systems without changing the kernel of the operating system. It consists of three major components: a core kernel module, a set of scheduler plug-ins and a library for the user space programs. As we will use ExSched in our implementation, brief descriptions of its different components are given in the following sections.

3.2.1 ExSched Core Module

The key component of the ExSched framework is its core module. It is a character-device module which can be loaded into kernels which support loadable modules. The core module is accessed by the user space programs through I/O system calls such as `ioctl()`. To determine scheduling decisions for the user program, the core module invokes a set of callback functions implemented by the specific scheduler plug-in. Finally the core module implements custom scheduling decisions from the plug-ins via original scheduling primitives of the hosting operating system.

For example, in Linux, the ExSched core uses `SCHED_FIFO` policy of the real-time scheduling class `rt_sched_class` to implement scheduling decisions from the plug-ins. It uses scheduling functions provided by the Linux kernel such as `schedule()`, `sched_setscheduler(task, policy,prio)` and `set_cpus_allowed_ptr(task, cpumask)` to implement real-time scheduling. In case of multiprocessors, task migration is done in two ways. The `migrate_task (task,cpu)` function of the core can migrate a task running in the thread context by simply calling the `set_cpus_allowed_ptr`. However, in case of task running in the interrupt context, the core module has to create a high priority real-time kernel thread to migrate the task.

To manage real-time properties, the core module has its own task descriptors with additional fields such as release time, deadline, etc. The original Linux task descriptor is part of the core task descriptor and plug-ins can also extend it via their own task descriptors. Linux bitmap queues are used to handle any task queues. The core module uses the kernel timer functions such as `setup_timer` and `mod_timer` to manage timing events. Three main internal functions namely `job_release()`,`job_complete()` and `rt_run_internal()` are used to provide interface to the scheduler plug-in to implement its scheduling policy via the core module.

3.2.2 ExSched User Space Library

ExSched framework provides a set of API functions for the real-time tasks to run using the core module. These functions are very simple and independent of underlying scheduling mechanisms. For example, `rt_enter()` and `rt_exit()` can be used to switch a normal task to real-time task and vice versa. `rt_set_priority(priority)` can be used to assign a priority to a task, while there are other functions to set the period, deadline, etc. A task can run in timeout mode using `rt_run(timeout)` and also can wait using `rt_wait_for_period()`. To summarize, this library provides a convenient way to write real-time tasks using ExSched.

3.2.3 ExSched Plug-in Developement

ExSched framework supports development of different schedulers via plug-in. To develop a plug-in in ExSched, the designer has to implement a set of callback functions that will be used by the core module. These are linked via the core module function pointers:

- `task_run_plugin`: this pointer links the plug-in function responsible for running the task.
- `task_exit_plugin`: this pointer links the plug-in function responsible for removing a real-time task from scheduling.
- `job_release_plugin`: this pointer links the plug-in function used in releasing task jobs.
- `job_complete_plugin`. this pointer links the plug-in function used when a task job is finished.

To illustrate how this framework can be used, we now explain two examples from the ExSched developers. These are the uniprocessor hierarchical EDF scheduling plug-in (`H-EDF`) and the semi-partitioned mulitprocessor scheduling plug-in (`FP-PM`).

In the `H-EDF` plug-in, the original data structures of the ExSched framework are extended to support hierarchical scheduling. This extension includes a new descriptor for the servers (`server_struct`), new queue management (`relPq`) to support both server and task queues and extension of original task descriptor of ExSched to attach it to a server. Besides new task run, job release and job complete handlers, the `H-EDF` plug-in also implements the server release and the server complete handlers. New handlers related to task jobs are modified to take the running server into account. The handlers for server release and server exit are complex as they handle both the server and tasks running using its budget.

In the `FP-PM` plug-in, very few new extensions of ExSched data structure is required compared to `H-EDF`. The only major modification is to add information related to task migration in the task descriptor. As this is a multiprocessor algorithm, handlers for running the task perform most of the complex computation required to decide which processor will run the task. Task migration is handled by the task run and the job complete handlers. However, all the handlers have to carefully manage time as tasks can migrate from one processor to another.

Chapter 4

Design

In this chapter we present our design of a virtual clustered hierarchical scheduler using ExSched. We intend to call our implementation as Virtual-Clustered Hierarchical Scheduling Framework (VC-HSF). First we present the system model for VC-HSF, then the design issues related to its implementation. Finally, we present the detail design of VC-HSF using ExSched.

4.1 System Model

A system contains a set Γ of n periodic real-time tasks $\tau_1, \ldots, \tau_n$ and a set C of k virtual clusters $C_1, \ldots, C_k$. Each periodic task τ_i is characterised by its relative deadline D_i, worst-case-execution time E_i and period T_i. Each periodic task τ_i belongs to a virtual cluster C_j and has a unique priority i within its virtual cluster. A virtual cluster C_j is characterised by its period Π_j, an execution budget per period θ_j, a concurrency parameter m'_j and a priority s_j. Each virtual cluster has a set of periodic tasks Γ_j which is a disjoint subset of Γ. Virtual cluster C_j can execute tasks of Γ_j using m'_j periodic servers $PS^j_1, \ldots, PS^j_{m'}$. A time-driven periodic server PS^j_i of virtual cluster C_j is characterised by a period Π_j, an execution budget Q^j_i and priority of its cluster s_j. The periodic servers idle their budget if there is no active task running in the server.

4.2 Design Issues

VC-HSF is based on VC framework presented by Shin et al. [50]. Although the authors gave an extended account on how VC framework may work [25], there is no guideline of how to implement it on a real operating system. Moreover recently, Lipari and Bini [41] pointed out implicit assumption in the VC framework that all servers are synchronized at the beginning of their period is hard to achieve in a multiprocessor environment and the proposed MPR model does not have a worst-case situation. As a result the schedulability analysis presented in the VC framework needs correction in estimation of worst-case supply. However, this limitation is relaxed in [35]. Another limitation of the VC framework noted in the survey by Davis and Burns [22] is that the number of task migration it requires may not be suitable for hard real-time systems.

However, the focus of our work is to implement the scheduling part of the VC framework in Linux. We don't intend to correct or extend the schedulability analysis of the VC framework, rather we want to implement and demonstrate its functionality which may give us valuable insights. In [55], the authors mentioned there are several multiprocessor architectures (several x86-32, ARM and UltraSPARC64) already available with a global clock for all cores. We also found these architectures are used in similar projects like $LITMUS^{RT}$ [17] where they used a global clock signal that is accessible to all processors in the form of cycle counter registers. According to those implementations, as Linux bases its notion of time on this global clock source, there is no drift among processors. Therefore synchronization of servers in different processors is not a big hurdle in VC implementation.

As mentioned earlier, we will use the ExSched [9] scheduler framework to develop a plug-in for VC-HSF. As a result, all subsequent discussions will be based on implementation support already provided by the ExSched.

Before proceeding towards the design of a VC-HSF multiprocessor scheduler, we have to take decisions about several design choices. We try to categorize them as below:

1. How to map tasks to the clusters?
2. How to map tasks to the periodic servers of the clusters?
3. How to assign the server budget to the servers of a cluster?
4. How can the schedulers at different level interact with each other?
5. How to synchronize the time events in different processors or how can we manage time?
6. How to capture real-time properties of tasks in the task, server and cluster level data structures? How to implement and manage queues of tasks? How many types of queues do we need?
7. How can we migrate tasks?
8. How to protect data structures that are shared by different handlers?

Now we will present some design options available for solving these queries in more detail.

4.2.1 Task to Cluster Mapping - Issue 1

In the description of the VC framework [50], [25] there is no clear guideline of how to map a task set into several clusters. However, Easwaran et al. [25] presented a theoretical analysis of two approaches as examples. In the first approach VC-Implicit Deadline Tasks (VC-IDT), each task is assigned to its own cluster having a single processor. This trivial approach is similar to the semi-partitioned and the partitioned EDF algorithms. The second approach mentioned is based on utilization separation, where tasks with high utilization have their own cluster and all other low utilization tasks are placed in a single cluster. However, both of these approaches suffer from large number of possible task preemptions which is inferior to non-cluster based approaches.

In [49], tasks are grouped into cluster based on the period-aware task allocation scheme. Their main idea was to put tasks which have harmonic periods into same cluster. This period-aware task allocation is shown [49] to incur low scheduler overhead if Boundary Fair (BF) algorithm is used for scheduling the clusters. Our observation is that, this approach can be also helpful for us as VC uses the Greatest Common Divisor (GCD) of the task periods to determine the period of MPR interface of a cluster. For example, using this scheme the task set of Figure 2.3 can be divided into two clusters having periods 2 and 3, each of which can be served by two periodic servers.

However, we can conclude that there is no concrete technique proposed so far for task to virtual cluster mapping and choice is open to subsystem designer. Clustering can be useful in placing restrictions on the amount of concurrency that a group of tasks can have. For example, suppose m tasks can thrash a L2 cache when they run concurrently. This problem can be solved by placing these m tasks into a virtual cluster having concurrency at most m' where $m' < m$. As a result, m tasks can not run in parallel and thus prevent them from thrashing the cache. In our implementation of VC-HSF we assume that the subsystem designer will specify the cluster affiliation of a real-time task during its creation. The subsystem designer can compose different subsystems as clusters from a system.

4.2.2 Task to Server Mapping - Issue 2

At this point of our analysis, we want to distinguish general two-level hierarchical scheduling from the virtual cluster-based hierarchical scheduling. In two-level hierarchical scheduling, each of the subsystem can be represented by a server which ensures temporal isolation between different subsystems. However in cluster-based scheduling, subsystems are represented by clusters which can be either physical or virtual. The most distinct difference introduced due to clustered subsystem is that unlike servers a cluster can simultaneously use more than one processor. As a result, in VC-HSF framework each of the clusters can be served by multiple servers, maximum number of servers that can be used simultaneously is bounded by the concurrency parameter of the cluster's MPR interface (m'). The servers within a cluster share task queues of their cluster. Therefore, there is no need to map tasks of a cluster to its servers.

4.2.3 Server Budget Assignment - Issue 3

In [25], budgets are assigned statically to servers either for its whole period (full budget) or remaining budget of the cluster in case the server is the last one belonging to that cluster. However, in our case, we use a more general model of budget assignment. We assign budget to the servers of a cluster when the cluster is released each time. A budget assignment function `assign_server_budget` is called each time the server is released to update the budget of the servers. This gives the flexibility of using different budget assignment policies in our implementation.

Different static budget assignment policies have different consequences on the schedule. For example, if full budget servers are used, these servers tend to occupy CPUs all the time as their budget will get replenished at the same time when it will get expired. As a result, sharing of CPUs can only happen in the CPU where servers running without full budget. In contrast, server budgets can be assigned completely dynamically at each scheduling point. In it if only one server is running, then it will receive all the budget from its cluster. When a new server of the same cluster is released, the remaining budget of the cluster should be distributed between the servers. However, this dynamic approach is complicated to implement in run-time.

For the implementation in this thesis we used a simple budget assignment policy that is very similar to balanced platform of a flexible interface as proposed in [35]. The algorithm works as follows: for m servers with total cluster budget Q, we assign $\lfloor \frac{Q}{m} \rfloor + 1$ budget to all servers except the last server, which will be assigned the remaining budget of the cluster. For example, total budget of 20 will be assigned as $(11,9)$ in 2 servers, where it will be assigned $(7,7,6)$ in 3 servers.

4.2.4 Schedulers - Issue 4

As discussed in the previous Section, the VC-HSF framework represents subsystems as clusters of tasks. There are two levels of scheduling described in VC, namely inter-cluster scheduling and intra-cluster scheduling. Here the inter-cluster scheduler refers to the global scheduler of the hierarchical scheduling while the notion of intra-cluster scheduler is same as the local scheduler.

In conventional hierarchical scheduling, the global scheduler schedules the servers representing the subsystem. The same is true for the inter-cluster scheduler of VC-HSF except that now each of the subsystem can have multiple servers. All the server tasks from all the clusters are queued according to any global scheduling policy and the single inter-cluster scheduler does this:

1. It is invoked whenever either a server budget is expired or get replenished via activation.
2. If m processors are available in a scheduling point, it picks m most suitable (such as highest priority ones in priority based scheduling) servers to execute on processes.
3. At each scheduling point, it has to check all the processors to find if there is a server running with lower priority that it can preempt. In case of server preemption, it should manage the remaining budget of the preempted server and insert it into the proper place of the server queue.
4. It invokes the intra-cluster scheduler whenever a server of that cluster either gets a new budget or gets preepmted or expired its budget.
5. It should manage status of the idle processors and try to schedule the preempted servers either on a idle processor or on any processor running lower priority server.

It is evident that the inter-cluster scheduler of the VC-HSF framework is more complex than the global scheduler of the hierarchical scheduling. Now we will look into how intra-cluster scheduler works.

The intra-cluster scheduler in the VC-HSF framework is responsible for scheduling tasks within the cluster into available processors for the cluster. It manages the real-time tasks similar to the local scheduler but uses the task queues of the cluster. However, the intra-cluster scheduler employs any global scheduling algorithm to schedule tasks into processors as there can be more than one processors available for the cluster. Processing time received by the cluster is represented by the periodic servers. This causes a new decision problem that needs to be resolved. Here we present the following two situations:

Option 1: Tasks are not mapped to the servers. As a result, whenever two servers get activated simultaneously the intra-cluster scheduler has to take the decision of which task to run on which processor. One thing to note here is that although all the servers of a cluster have the same period, their individual budgets may differ. Therefore it can happen that the scheduler schedules a longer running task into a server with shorter budget when two servers with different budget gets activated at the same time. This can cause unnecessary preemptions which can be avoided if the intra-cluster scheduler takes server budget into account while allocating tasks to the processor (see Figure4.1). However, checking server budget while allocating tasks to that servers processor will make the intra-cluster scheduler more complex.

Option 2: Tasks are mapped to a particular server of its cluster by the system designer. This makes local scheduling simpler with the expense of additional data structure related cost of task-to-server mapping. However, this idea is not presented in the original VC framework and there is no guideline for it. So, to implement this option we need to propose additional methods of task-to-server mapping in our VC-HSF.

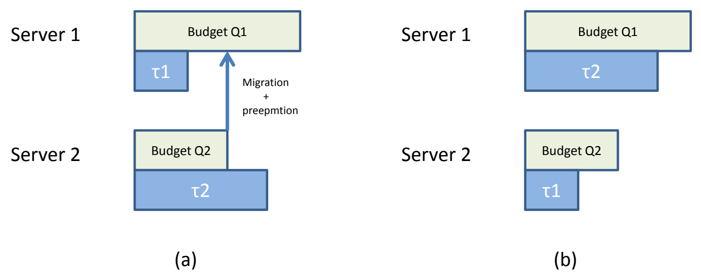

Figure 4.1: Intra-cluster scheduling problem with no task-to-server mapping (a) unnecessary migration (b) better choice by the scheduler

Both of the above mentioned problems require additional optimization of the original VC framework. To simplify our initial implementation we adhere to the assumption in option 1 and allow intra-cluster scheduler to run a ready task in any of the available server of its cluster.

The VC-HSF framework needs one intra-cluster (local) scheduler per cluster instead of one local scheduler per server of the hierarchical scheduling. All the servers in a single cluster share task queues of that cluster and the inter-cluster scheduler works in the following way:

1. It is invoked by the inter-cluster scheduler whenever one of its server either receives or expires its server budget or gets preempted by the other servers. It is also invoked when a new task job of the cluster becomes ready or completes its execution.

2. If n servers are available at a scheduling point, it picks n most suitable (such as highest priority ones in priority based scheduling) tasks to execute on them.

3. It inserts or removes the task job into the runqueues of the assigned processor via the ExSched core module functions.

4. It migrates its task to a new processor whenever server running that task is migrated by the inter-cluster scheduler.

In the ExSched framework, schedulers are implemented by scheduler plug-in functions that are fired when a scheduling event relevant to that function occurs. The ExSched framework manages real-time task execution by the core module which uses Linux queues to keep all the task structures. However, our VC-HSF framework needs cluster wise task queues and requires its manipulation only by our defined plug-in functions. In addition, we implement our schedulers as task, server and cluster level handler functions similar to other plug-in development of ExSched. Unlike the other multi-core scheduling plug-ins of ExSched which uses core-wise kernel threads, all components of our inter-cluster and intra-cluster scheduler is placed in a single CPU which also contains all the global data structures. This approach is taken to simplify the scheduler development and synchronization between different scheduler handlers.

For the simplicity of the implementation, we use Fixed Priority (FP) scheduling in both inter-cluster and intra-cluster scheduling. As a consequence, we assign static fixed priority to our clusters and tasks. Server of a cluster inherits priority of its cluster.

4.2.5 Time Management - Issue 5

The ExSched framework supports event-based scheduler development. In case of Linux, time granularity used for this event-based scheduling is the global variable `jiffies`. Our implementation also uses jiffies as the granularity of time. Default `jiffies` frequency is 100 Hz and 250 Hz (recent versions), which gives tick value of 10 ms and 4 ms respectively. However, we recompiled our implementation kernel with 1000 Hz `jiffies` frequency and get tick value of 1 ms. Although `jiffies` is a global variable of the operating system valid for all CPUs running under it, individual CPUs may not be in finer resolution synchronized in time. In our experience, the effect of this timing difference is not visible upto 1 ms tick level. As a result, we are not using any special procedure to synchronize time in all processors, rather relying on the global `jiffies` to determine time events.

As we have mentioned in the previous section, ExSched scheduler development requires implementation of a number of plug-in functions to handle the scheduling events. Timing events are handled by timer interrupts which calls the handler function when the timer expires. The Linux kernel provides `timerlist` timers to defined interrupts generated by timing events. We summarize the timing events in the VC-HSF framework as in Table 4.1:

Table 4.1: Timing events in VC-HSF

Event	Scheduler invoked	Type
Cluster budget released	Inter-cluster	Global
Server budget expired	Inter-cluster	Per server
Task job released	Intra-cluster	Per task

As all task related timing events are handled by the ExSched core, we only need to handle two types of timing events; events of cluster budget release and server budget expiration. To simplify our design we use only two types of timers. We use one global timer for managing cluster release and one timer per server to manage its budget expiration. The global timer called `event_timer` is used only for cluster release events. We handle simultaneous release of the clusters in our cluster release handler. However the timers used by the servers and ExSched can expire at the same time as the `event_timer`. We need to handle such events by synchronizing handler functions used by the timers.

4.2.6 Data Structures - Issue 6

In the VC-HSF framework, we have three types of entities, namely task, server and cluster. As a result, we need atleast three different types of descriptors to keep all the relevant information about these entities. In addition, to manage the running status of the processors in our system we need additional data structure which should contain whether a server of a cluster is currently running on that CPU.

For scheduling, the scheduler needs to enqueue and dequeue the descriptors of clusters, servers and tasks into some queues. In the cluster level, all clusters waiting to be released should be placed in a release queue which is sorted based on absolute time of their release. When a cluster is released but not able to run, the scheduler should to keep it in a cluster ready queue where descriptors are sorted based on the inter-cluster scheduling policy. In case of servers, we use release of a cluster as release of all its servers. The advantage of this approach is that we do not need additional mechanism for releasing servers. However, all the servers of an active cluster may not have processors available to run. These servers which are ready to run are kept in a global server ready queue, sorted based on their priority. As the servers inherit the priority of their cluster, a server of higher priority cluster will be always in front of a server of lower priority cluster in the queue. In our implementation, we rely on a modified release mechanism of ExSched for releasing task jobs. In that case, we do not require any release queue for the tasks. However, the released task jobs should be enqueued in a task queue of its cluster. This queue is similar to the ready queues of clusters and servers, but sorted based on the task priority. Single queue for all released job per cluster ensures that tasks of same cluster only compete between each other to use the cluster budget.

As we see the scheduling mechanism requires a large number of queue operations, the efficiency of the scheduler largely depends on how we implement the queues for them. Following original Linux task runqueue implementation and its successful use in the HSF implementation [9], we use bitmap queues in all of our queue implementation. A bitmap queue consists of a bitmap which is an integer variable and indexed array of linked lists (see Figure4.2). Each bit of the bitmap represents an index of the array of linked lists which may represent different priorities. A set bit means the corresponding linked list is not empty, a zero bit represents there is no descriptors in the list indexed by the bit number. The advantage of using this queue is that, the queue lookups are very fast using bitwise operations of *C* such as Find First Set Bit (`FFS`) or Find Last Set Bit (`FLS`). Only limitation of this kind of queue is that size of the bitmap should be power of two, which also depends on size of the variable used for representing the bitmap. For example, if we use simple integer, number of priority levels will be 32. In case we want to implement a release queue based on time, this imposes requirement of handling wrapping around of bitmap levels. However, due to fast queue operations provided by this type of queues, we use them in all of our queue implementation.

4.2.7 Migration of Tasks - Issue 7

In ExSched, migration of tasks is performed in two ways. The first way is to simply configuring the `cpu_mask` of the Linux task and then calling the `set_cpus_allowed_ptr`. However, this migration procedure calls the Linux `schedule()` function which is not allowed to be called from an interrupt handler. This is only allowed for fully preemptible kernel with `CONFIG_PREEMPT` option enabled. So, to use task migration from interrupt handlers, ExSched uses migration threads. A migration thread is a kernel thread with highest priority which can migrate tasks when the scheduler interrupt handler finishes its execution. Implementation of migration thread depends on how global scheduling is implemented using ExSched. To simplify our implementation, we are using the first method of preemptible kernel. We compiled our working

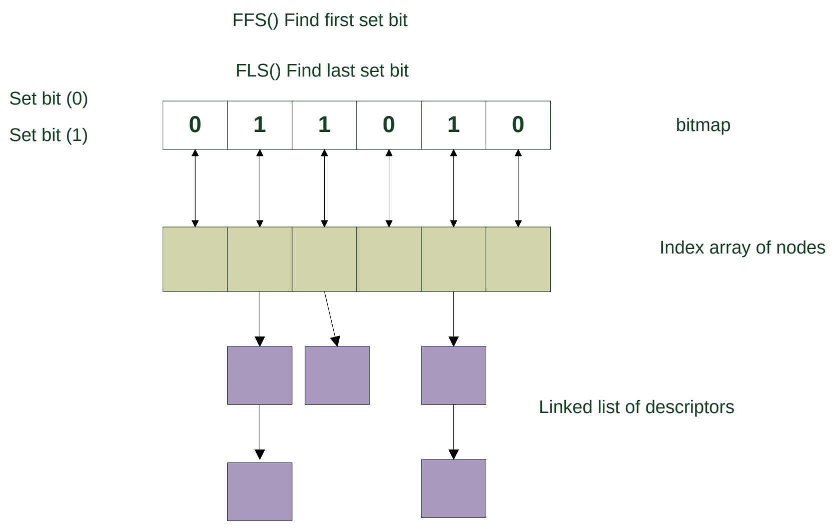

Figure 4.2: Structure of a bitmap queue

kernel with `CONFIG_PREEMPT` option which allows calling `schedule()` even from inside of a interrupt handler. However, with synchronization between interrupt handlers for global data structure access in our implementation, `schedule()` is called always after completion of an interrupt handler. This is explained in the Section 4.1.8.

4.2.8 Synchronization of Access to Global Data Structures - Issue 8

As we have seen in the previous sections, in VC-HSF most of the queues and other data structures are global. Different interrupt handlers (cluster, server or task level) should access these data structures without race conditions. If the access to these global data structures are not protected with locks then there can be inconsistent states which can result in a crash of the operating system.

To protect the access to global data structure we use the native Linux spinlocks. The spinlocks are simplified lock mechanism where the code trying to get a lock which is already held by some other code will continue busy-waiting in the point of acquiring lock. As our implementation is in multi-core, we use the recommended version for the SMP lock, which is the `spin_lock_irqsave` and `spin_unlock_irqrestore`. To simplify the implementation, we use coarse-grained locking, which means we protect the whole code of the function by taking the spinlock at the beginning of the handler code and releasing at the end. This simple approach gives protection over all critical and non-critical code inside our interrupt handlers which accesses global data structures.

As mentioned earlier, we compiled our kernel with `CONFIG_PREEMPT` to simplify task migration mechanism. This have an interesting consequence together with the use of spinlocks. With `CONFIG_PREEMPT` enabled, the whole kernel code is preemptible which means `schedule()` should be called whenever the task migration is done, even from inside of a interrupt handler. However, when `CONFIG_PREEMPT` is enabled, spinlocks simply becomes preemption disable and enable option for the portion of the protected code. As a result, even with `CONFIG_PREEMPT` enabled, `schedule()` can be only called when our interrupt handler releases the spinlock.

4.3 Detail Design

Following the discussions presented in the previous sections, now we will explain our design in details.

4.3.1 Extension of ExSched

In original ExSched, tasks are completely managed by the core module and there is no support for clusters. We removed these limitations by following extensions:

1. We extended the original `resch_task_t` data structure (task structure of ExSched) with a field donating cluster_id of the task.
2. To assign an ExSched task to a cluster, we added an API function called `api_set_cluster`.
3. We introduced a new macro `RESCH_VCHSF` in ExSched core. When this macro is defined, it will disable all default code portions of ExSched which handles task queuing and execution by the core.

4.3.2 Task Execution

For running a VC-HSF task using ExSched, we utilize three plug-in function calls defined by:

- VC-HSF task run plug-in function is called from the `api_run` function call of the real-time task.
- ExSched job release function should forward all responsibility of a newly released job to VC-HSF job release plug-in function.
- When a job is finished by calling `api_wait_for_next_period()`, the VC-HSF job complete plug-in function should handle the completed job.

Figure 4.3 shows how a VC-HSF task executes using the ExSched framework.

4.3.3 Descriptors

There are three main types of descriptors required in the VC-HSF scheduling framework. These are task descriptor, server descriptor and cluster descriptor.

Task descriptor: The original task descriptor of ExSched framework defined by the structure `resch_task_t` is designed with real-time tasks in mind. It has all the necessary properties related to a general task (such as id, pid, state, timer, etc.) with additional properties for defining a real-time task (such as priority, period, relative deadline, execution time, etc.). It also includes processor specific information via `cpu_mask` and `cpu_id` fields. However, this default task descriptor does not capture task migration and cluster related information. To support these we need a new task descriptor called `vctask_t` on top of default `resch_task_t`.

In our design, we intend to keep the new `vctask_t` data structure as light-weight as possible. We need only three fields in `vctask_t`:

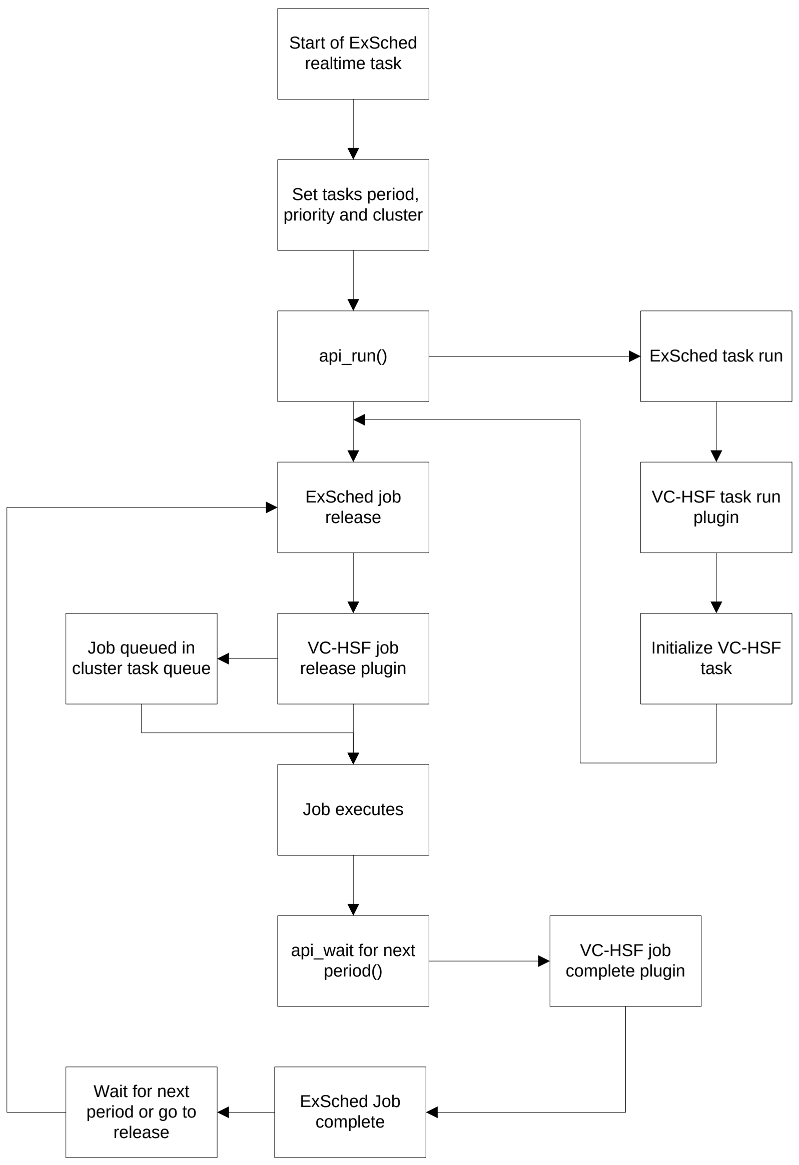
Start of ExSched realtime task
Set tasks period, priority and cluster
api_run()
ExSched task run
ExSched job release
VC-HSF task run plugin
Job queued in cluster task queue
VC-HSF job release plugin
Initialize VC-HSF task
Job executes
api_wait for next period()
VC-HSF job complete plugin
Wait for next period or go to release
ExSched Job complete

Figure 4.3: Execution of real-time task in VC-HSF

- `resch_task_t` rt: This is a pointer to the `resch_task_t` of the ExSched task.
- `priority`: This field represents the priority of the ExSched task and used in task queues.
- `vctask_t` next: The next pointer to be used in the task queues.

Server descriptor: Each of the servers in a cluster has its own server descriptor defined by the structure `vc_server_t`. `vc_server_t` has the following important fields:

- `vc_server_id`: This is the identifier for denoting a server.
- `vc_server_cpu`: This indicates the `cpu_id` of the processor where the server is running. The inter-cluster scheduler uses this to assign the server to processor.

- `vc_server_period`: This is the period of the server denoted by Π parameter of the MPR interface.
- `vc_server_priority`: This is the priority of the server used in inter-cluster scheduling. In our case, we assign a server to priority of its cluster.
- `vc_budget_expiration_time`: The time when the budget of the server should expire.
- `vc_server_budget`: This field denotes the total server budget assigned to this server.
- `vc_server_remain_budget`: This field is used to denote the remaining budget of a server when it gets preempted.
- `vc_mycluster`: This is a pointer to the cluster descriptor in which this server belongs to. Intra-cluster scheduling can use this field to access task queues of the cluster and thus run a task using this server.
- `server_budget_timer`: This is a timer that is used to detect the budget expiration when the server is running.
- `active_task`: This is a pointer to the task structure of the currently running task using the budget of this server.
- `running`: A flag value to indicate whether the server is running or not.
- `timestamp`: This field is the timestamp that is used in case of server preempetion to calculate the remaining budget of the server.
- `next`: A pointer to the next element in the queue.

Cluster descriptor: Each of the clusters in the system has a cluster descriptor defined by the structure `vc_cluster_t`. `vc_server_t` has following fields:

- `vc_cluster_id`: This is the unique identifier for denoting a cluster.
- `vc_cluster_state`: This field indicates the state of the cluster. A cluster can have one of these three states: (1) `CLUSTER_READY`: means the cluster is released but none of its server is released. In this case, the cluster is waiting in a queue, (2) `CLUSTER_ACTIVE`: means the cluster is released and at least one of its servers has executed its budget and (3)`CLUSTER_EXPIRED`: means the cluster expired its budget in the current period and should not run any more servers.
- `vc_ncpus`: This indicates the maximum number of CPUs that are allowed to execute concurrently for this cluster, which is the m parameter of the MPR interface. This number also indicates the number of periodic servers used by the cluster.
- `vc_period`: This is the period of the virtual cluster denoted by Π parameter of the MPR interface.
- `vc_cluster_budget`: This field denotes the total execution requirement of all the tasks in the cluster equivalent to Θ parameter of the MPR interface.

- `vc_cluster_remain_budget`: This field is used to keep track of the remaining budget of the cluster, which is used to assign budget to the server of the cluster.
- `nr_active_servers`: This fields denotes how many servers of this cluster are running.
- `nr_ready_servers`: This fields denotes how many servers of this cluster are ready, waiting to execute their budget.
- `SERVERS`: This is an array of `vc_server_t`s of the servers that belong to this cluster.
- `READY_TASKS`: This is the task priority queue of the cluster.

CPU status descriptor: To keep information about the state of the processors running the server and the tasks, we use a data structure called `cpu_resource_struct`. This data structure has the following fields:

- `cpu_busy`: This field has two states, `CPU_IDLE` will indicate no server is running in this CPU. On the other hand, `CPU_BUSY` will show the there is a server running in it.
- `active_server`: This is the pointer to the `vc_server_t` of the actively running server on this CPU. In case of `CPU_IDLE`, this should be NULL.
- `active_cluster_priority`: The priority of the cluster whose server is running in this CPU.

Figure 4.4 shows how different main descriptors of VC-HSF are interconnected with each other.

4.3.4 Queues

In our VC-HSF there are three different types of descriptors that require queuing mechanism (for clusters, servers and tasks). We will now explain how we designed these queues and how they work.

Firstly, as explained in the previous section, each of the clusters has their own cluster_t descriptor with priority, period and budget. Clusters are released with a new budget when their period expires and this is done using a release queue. However, it is possible that a cluster is released but none of the processors is available for it to run. In that case, we use another queue of cluster_t called cluster ready queue. In case, when a processor is available for that cluster to run it is removed from the cluster ready queue. These queues are implemented as following:

- CLUSTER_RELEASE_QUEUE: This queue is implemented as a bitmap queue as used by the original Linux. However, as it is a release queue we need to sort queue elements based on increasing order of time. Each index of the linked list array of this queue represents a single time unit, which in our case is a single `jiffies` or 1 ms. The queue is initialized with a bitmap size larger than the largest possible period defined for any cluster in the system. The queue implementation also needs to handle wrapping up of time values in bitmap. To handle the wrap up, we use two bitmaps. Whenever a bitmap wraps up we continue inserting the values to the second one. However to determine the exact position of a wrapped up index we need to use a variable called `virtual time` which is initialized

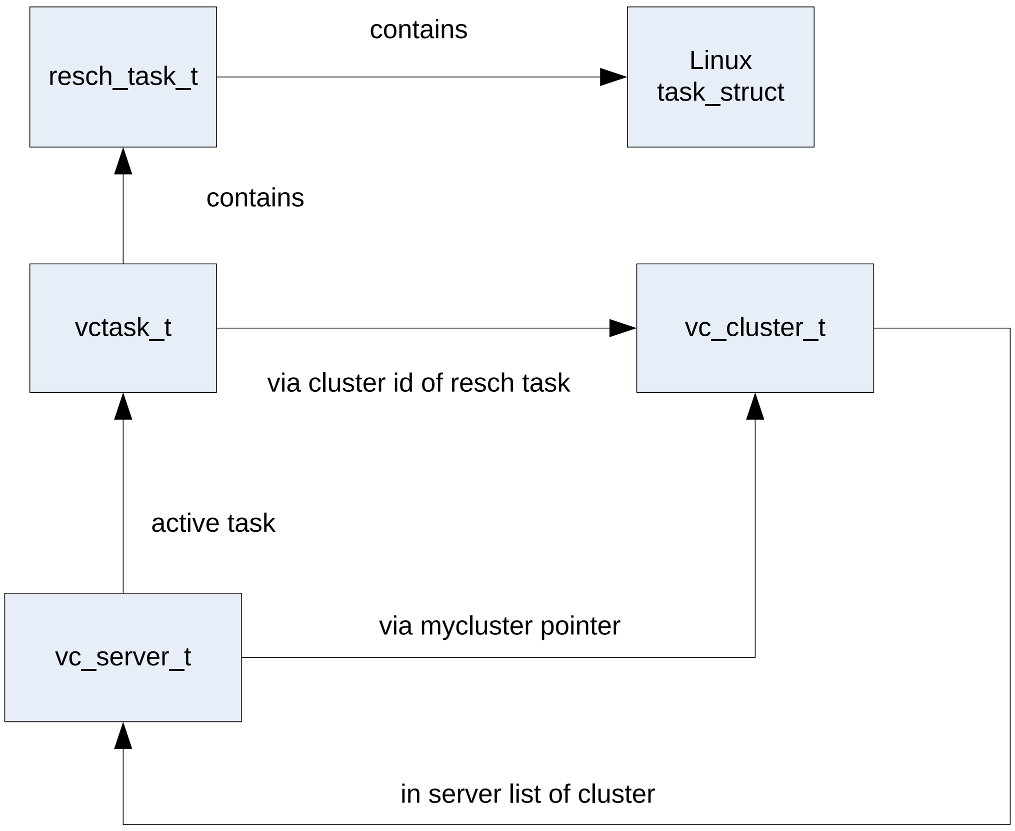

Figure 4.4: Overview of main descriptors in VC-HSF

when the release queue is initialized. This `virtual time` is modified whenever there is a release event and it is advanced to the next release event. Figure 4.5 shows with a simple example how the wrapping up of time value in bitmap release queue can be handled using `virtual time`. Any look up into the queue can be done in constant time because it is equivalent to finding the index of the lowest set bit using bitmap operation. The only limitation of this implementation is that size of the bitmap queue should be power of two. This queue is defined as type `relpq`.

- CLUSTER_READY_QUEUE: The ready queue is implemented as a bitmap queue, sorted on the priority values of the clusters that are ready. It is simpler than the release queue as there is no need to handle time. This queue is defined as type `pq`.

Secondly, servers of the clusters that are ready to run need a queue. For this reason we used a global ready queue for all the servers from all active servers. One thing to note here, by an active cluster we mean a cluster which already is able to run at least one of its server in its current period. This queue is:

- SERVER_READY_QUEUE: This ready queue is implemented as a bitmap queue same as the cluster ready queue. The only major difference is that the queue elements in it are all server descriptors. This queue is defined as type `spq`.

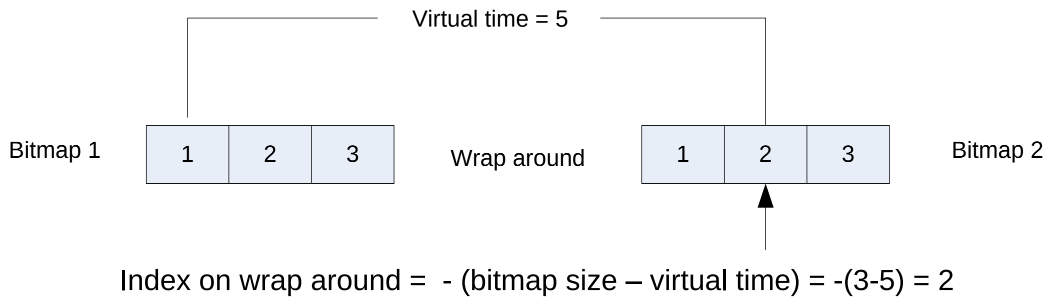
Virtual time = 5
Bitmap 1
1
2
3
Wrap around
1
2
3
Bitmap 2
Index on wrap around = - (bitmap size – virtual time) = -(3-5) = 2

Figure 4.5: Wrapup handling in release queue using `virtual time`

Finally, each of the clusters requires a queue for all the released but not running task jobs. This cluster specific queue is a part of the cluster_t descriptor. We define it as:

- READY_TASKS: This task ready queue is also a bitmap queue with `vctask_t` descriptors as elements. The queue is sorted based on task priority and defined as type `tpq`.

4.3.5 Scheduler Plug-in Functions

As described in the Section 4.3.2, VC-HSF needs to implement plug-in functions for job release and job complete. These two plug-in functions are called from the ExSched job release and job complete handlers.

Job Release Handler

The job release handler of VC-HSF tries to run the job when it is released. In case it is not successful, the job is enqueued into the task queue of the jobs cluster. Figure 4.6 shows operations of the VC-HSF job release handler.

Job Complete Handler

The job complete handler of VC-HSF simply tries to run a pending job of the same cluster when a job is finished. Figure 4.7 shows operations of the VC-HSF job complete handler.

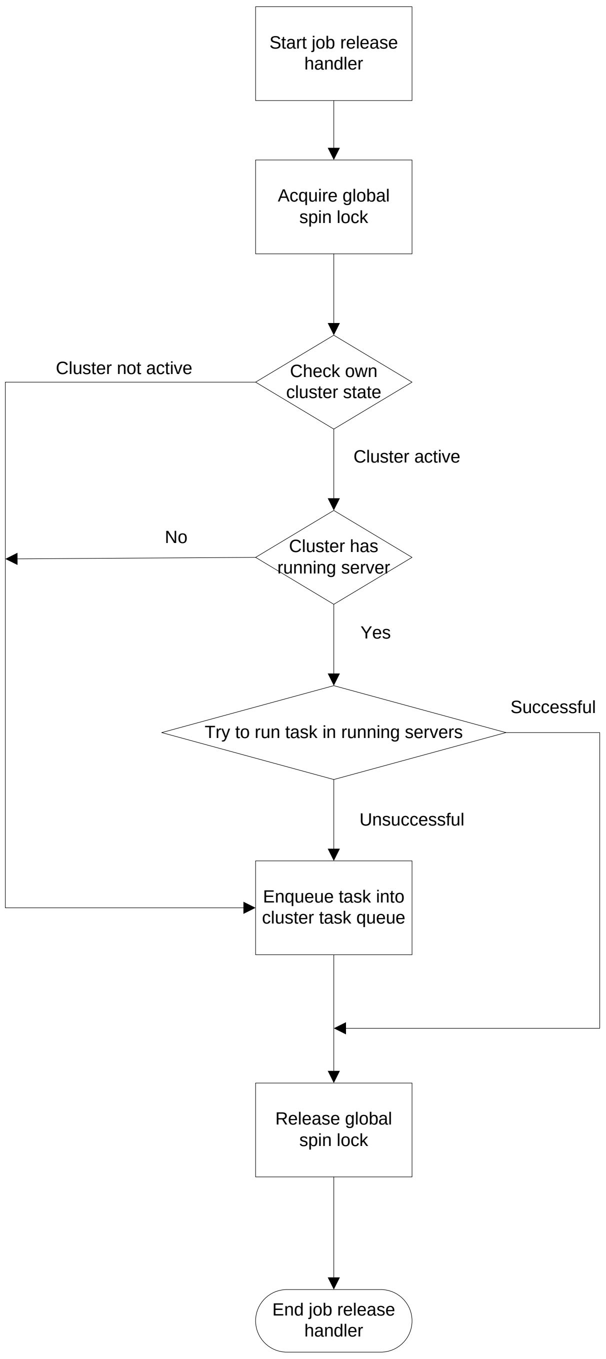

Figure 4.6: Operations of VC-HSF job release handler function

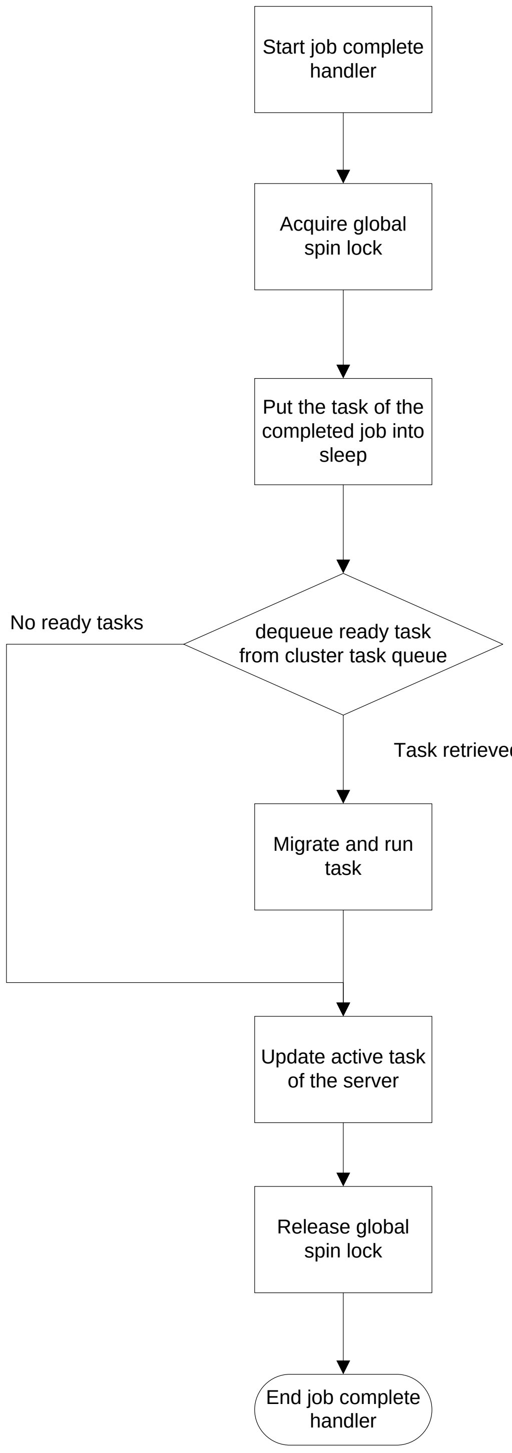
Start job complete handler
Acquire global spin lock
Put the task of the completed job into sleep
dequeue ready task from cluster task queue
No ready tasks
Task retrieved
Migrate and run task
Update active task of the server
Release global spin lock
End job complete handler

Figure 4.7: Operations of VC-HSF job complete handler function

4.3.6 Scheduler Interrupt Handlers

Cluster Release Handler

This interrupt handler is independent of ExSched. Its execution is divided into two main cases. In the first run, all clusters are ready to run, so no cluster needs to be released. In that case, this handler will launch clusters according their priority and available processors. The second or more general case requires release of either one or multiple cluster in a time instant. However, same as the first run it will try to run as more number of released clusters as possible. In the worst case, all released cluster at the release time will be able to run simultaneously. At the end, the handler needs to update the global `event_timer` to the nearest possible next release event. Different parts of the operations of the VC-HSF cluster release handler is shown in the Figures 4.8, 4.9, 4.10 and 4.11.

Server Complete Handler

This interrupt handler is called when the budget of a server expires. If the budget expired server has a running task, this task will try to migrate that task to another server of the same cluster which is running a lower priority task. If it is not successful than the running task of the expired server is enqueued back to task queue of the cluster. If the expiration of the budget causes all the budget of the servers cluster in the current period to expire, then this handler should call the cluster complete handler. Additionally when a server completes, this handler will first try to run the highest priority ready server in the idle cpu. If there is no such ready server, then the handler will try to launch ready clusters and their servers. Figure 4.12 shows operations of this server complete handler in more detail.

4.3.7 Major Non-Interrupt Functions

There are some important functions in VC-HSF which are not interrupt handler. These are:

1. Cluster complete handler: This function is called when the budget of a cluster expires. As this event can only happen when the budget of the last running server of the cluster expires, cluster complete handler is always called from a server complete handler. Figure 4.13 shows the operations of the VC-HSF cluster complete handler.

2. Try to run server: This function is always called when any of the interrupt handler tries to run a server. Figure 4.14 shows the operations of the try to run server function of VC-HSF.

3. Run server: This function is called from the try to run server to run a server in an idle processor. Figure 4.15 shows the operations of the run server function of VC-HSF.

4. Preempt server: This function is called from the try to run server when a running server needs to be preempted. Figure 4.16 shows the operations of the preempt server function of VC-HSF.

5. Try to run task: This function is called when the scheduler wants to run a VC-HSF task. Figure 4.17 shows the operations of the try to run task function of VC-HSF.

4.3.8 Miscellaneous Functions and Global Variables

There are few more important functions that are used frequently by all the functions presented above. These are:

- `migrate_vchsf_task`: This function migrates a task to a new cpu. It has three steps; first its clears the `cpu_mask` of the task, then set the bit of destination cpu on it and finally call the `set_cpus_allowed_ptr` function.
- `update_lowest_cluster_prio`: This function updates global variable `lowest_active _cluster_priortiy` by doing a linear search on the list of all cluster descriptors. This global variable is updated only inside handlers and the update function is called in the handlers when either a cluster is launched or complete.

There are two other important global variables which are updated carefully and used by the handlers as:

- `idle_cpus`: The number of cpus currently which are not running any server.
- `lowest_active_cluster_priortiy`: This value is useful when the server of a cluster tries to preemept another lower priority server to run in its cpu. To update it, scheduler has to do a linear search in the array of `vc_cluster_ts` to find out the lowest active priority one.

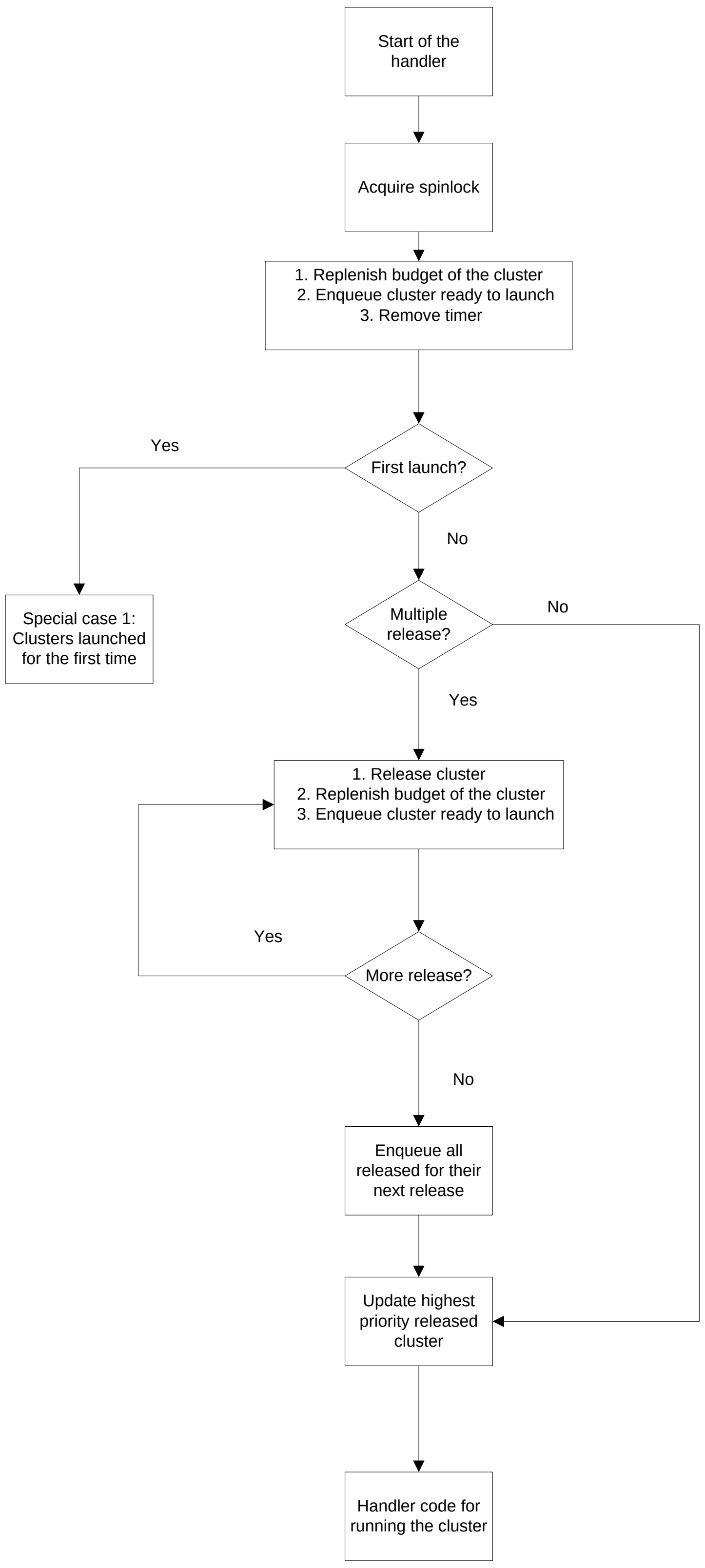
Start of the handler
Acquire spinlock
1. Replenish budget of the cluster
2. Enqueue cluster ready to launch
3. Remove timer
First launch?
Yes
No
Special case 1: Clusters launched for the first time
Multiple release?
No
Yes
1. Release cluster
2. Replenish budget of the cluster
3. Enqueue cluster ready to launch
More release?
Yes
No
Enqueue all released for their next release
Update highest priority released cluster
Handler code for running the cluster

Figure 4.8: Cluster release handler beginning operations

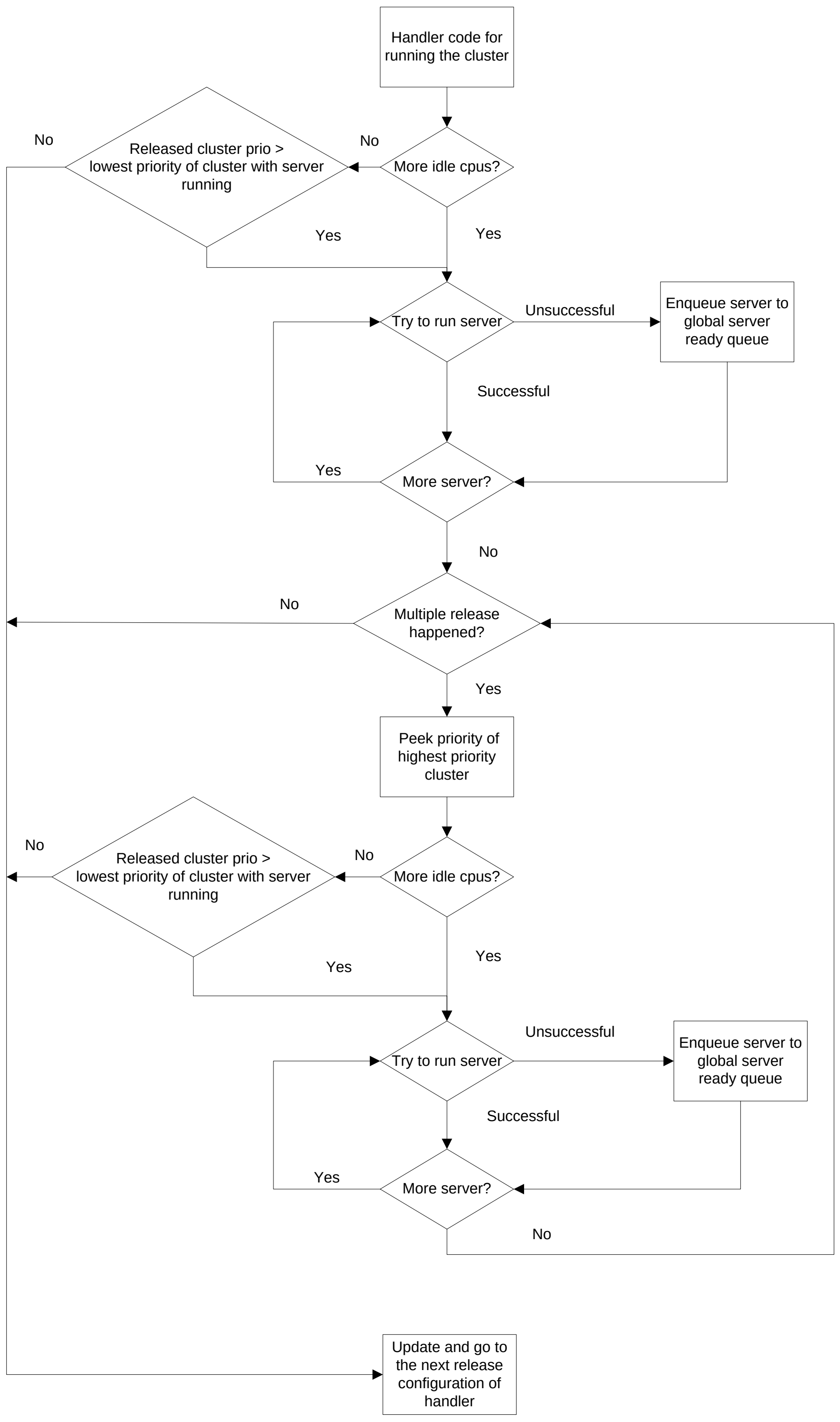
Handler code for running the cluster
More idle cpus?
No
Released cluster prio > lowest priority of cluster with server running
No
Yes
Yes
Try to run server
Unsuccessful
Enqueue server to global server ready queue
Successful
More server?
Yes
No
Multiple release happened?
No
Yes
Peek priority of highest priority cluster
More idle cpus?
No
Released cluster prio > lowest priority of cluster with server running
No
Yes
Yes
Try to run server
Unsuccessful
Enqueue server to global server ready queue
Successful
More server?
Yes
No
Update and go to the next release configuration of handler

Figure 4.9: Cluster release main operations

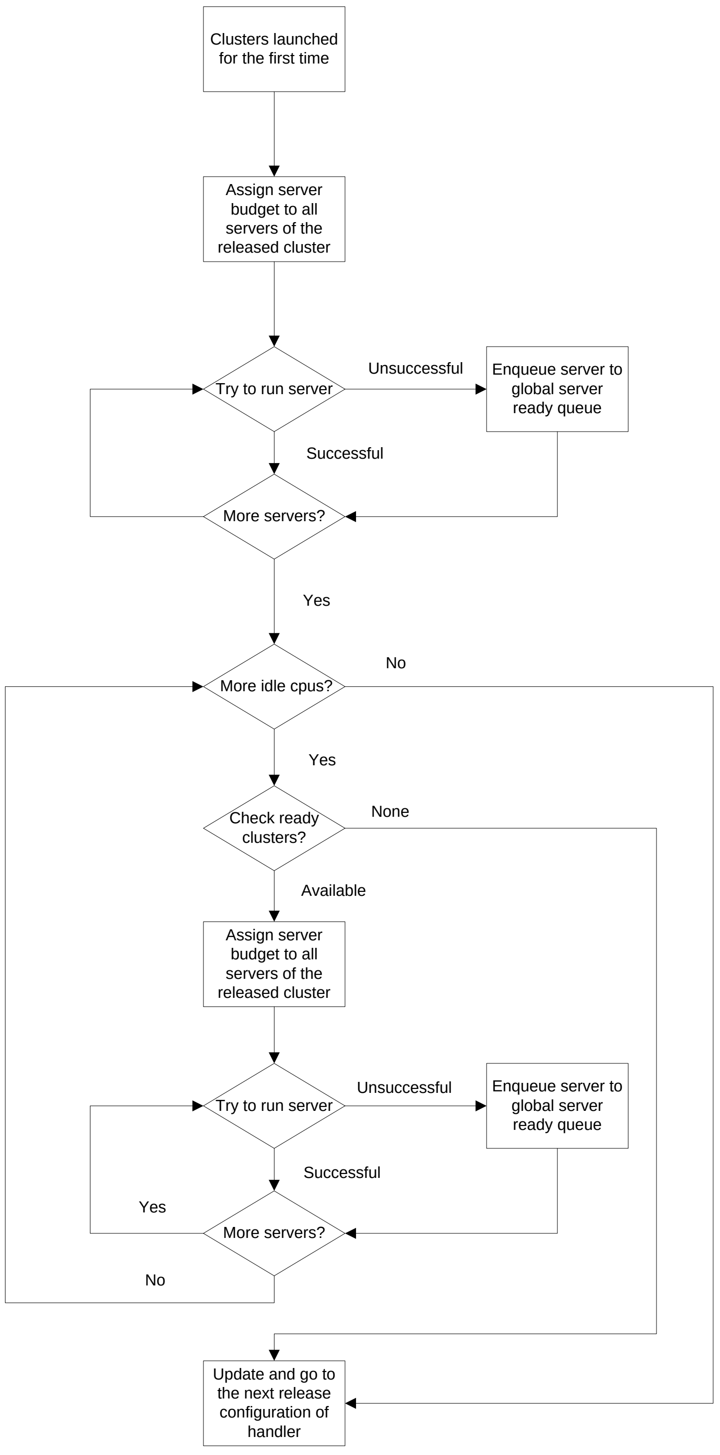
Clusters launched for the first time
Assign server budget to all servers of the released cluster
Try to run server
Unsuccessful
Enqueue server to global server ready queue
Successful
More servers?
Yes
No
More idle cpus?
Yes
Check ready clusters?
None
Available
Assign server budget to all servers of the released cluster
Try to run server
Unsuccessful
Enqueue server to global server ready queue
Successful
Yes
More servers?
No
Update and go to the next release configuration of handler

Figure 4.10: Cluster release handler first run operations

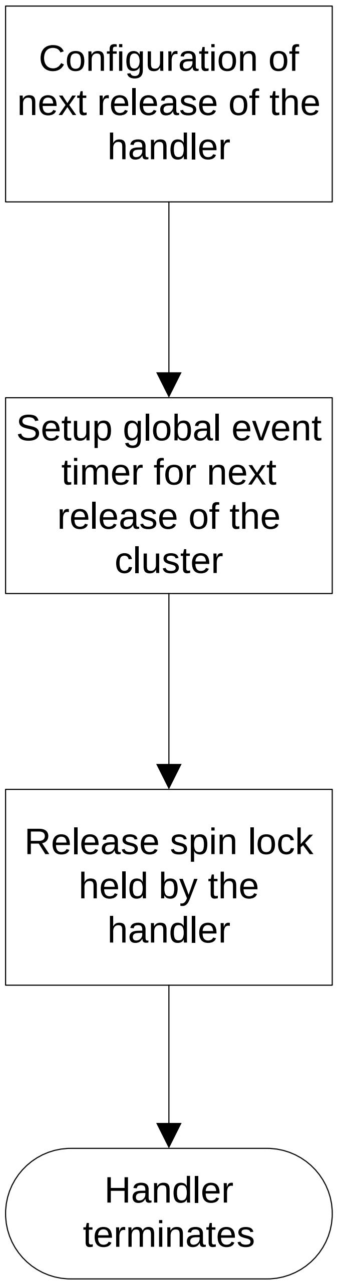
Configuration of next release of the handler
Setup global event timer for next release of the cluster
Release spin lock held by the handler
Handler terminates

Figure 4.11: Cluster release handler end operations

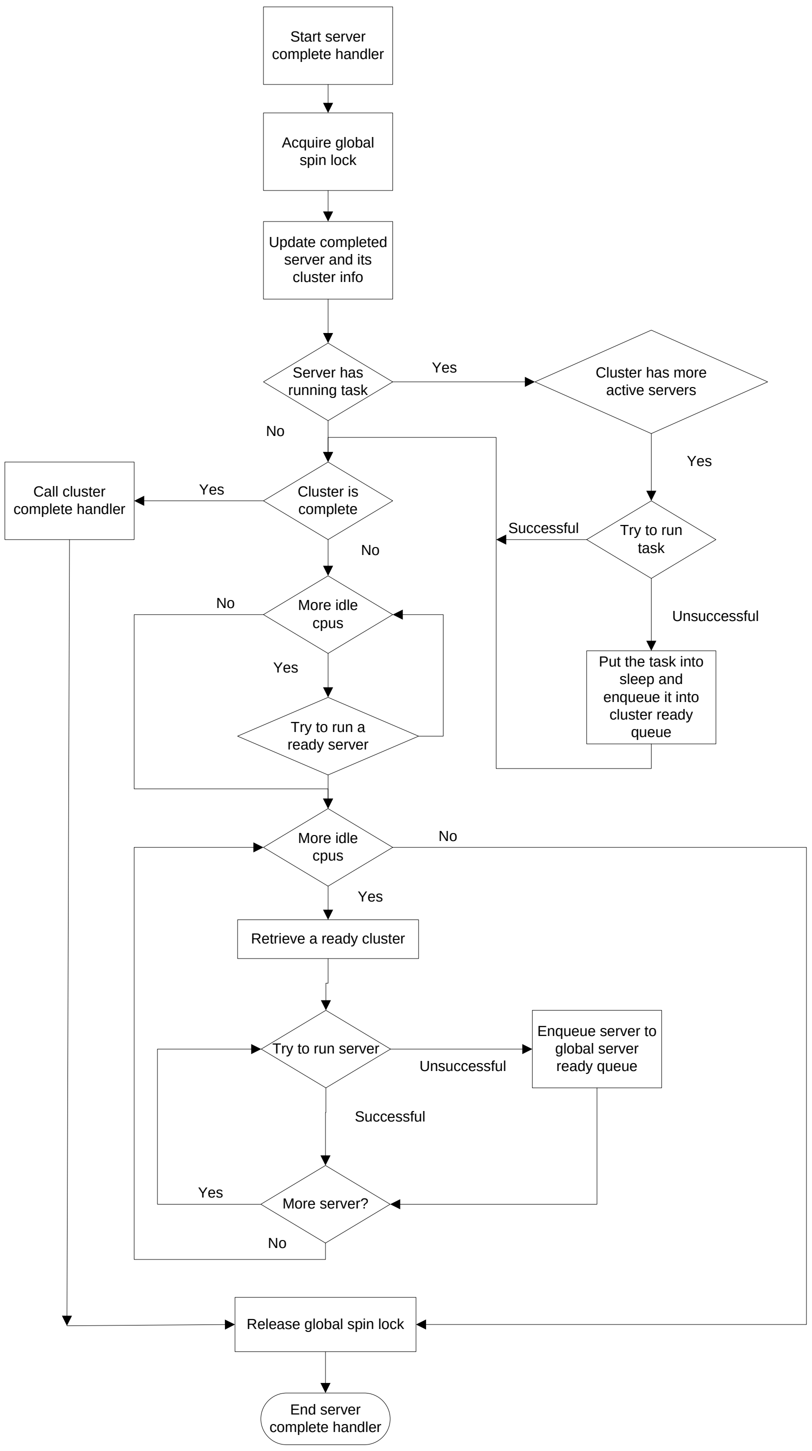
Start server complete handler
Acquire global spin lock
Update completed server and its cluster info
Server has running task
Yes
Cluster has more active servers
No
Yes
Call cluster complete handler
Yes
Cluster is complete
Successful
Try to run task
No
No
More idle cpus
Unsuccessful
Yes
Put the task into sleep and enqueue it into cluster ready queue
Try to run a ready server
More idle cpus
No
Yes
Retrieve a ready cluster
Try to run server
Unsuccessful
Enqueue server to global server ready queue
Successful
Yes
More server?
No
Release global spin lock
End server complete handler

Figure 4.12: Server complete handler operations

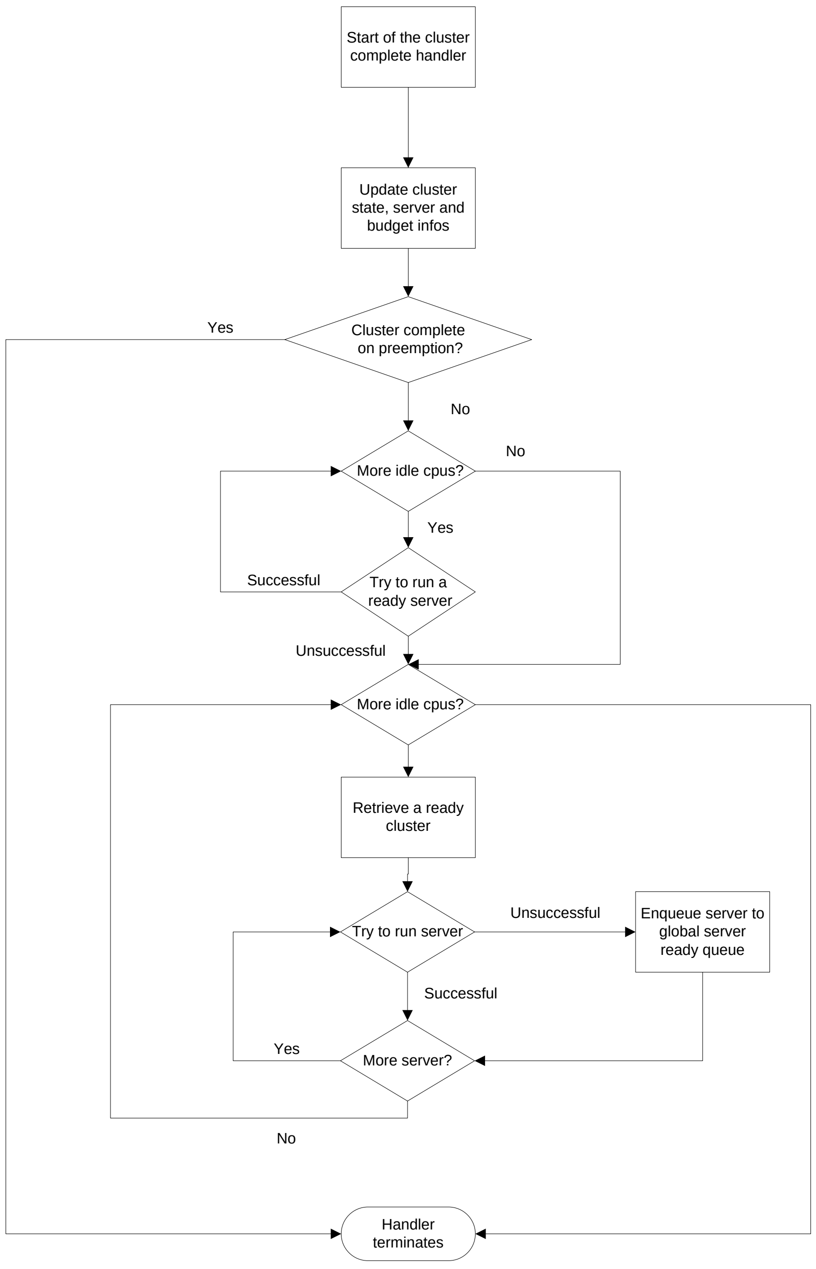
Start of the cluster complete handler
Update cluster state, server and budget infos
Cluster complete on preemption?
Yes
No
More idle cpus?
No
Yes
Successful
Try to run a ready server
Unsuccessful
More idle cpus?
Retrieve a ready cluster
Try to run server
Unsuccessful
Enqueue server to global server ready queue
Successful
Yes
More server?
No
Handler terminates

Figure 4.13: Cluster complete handler operations

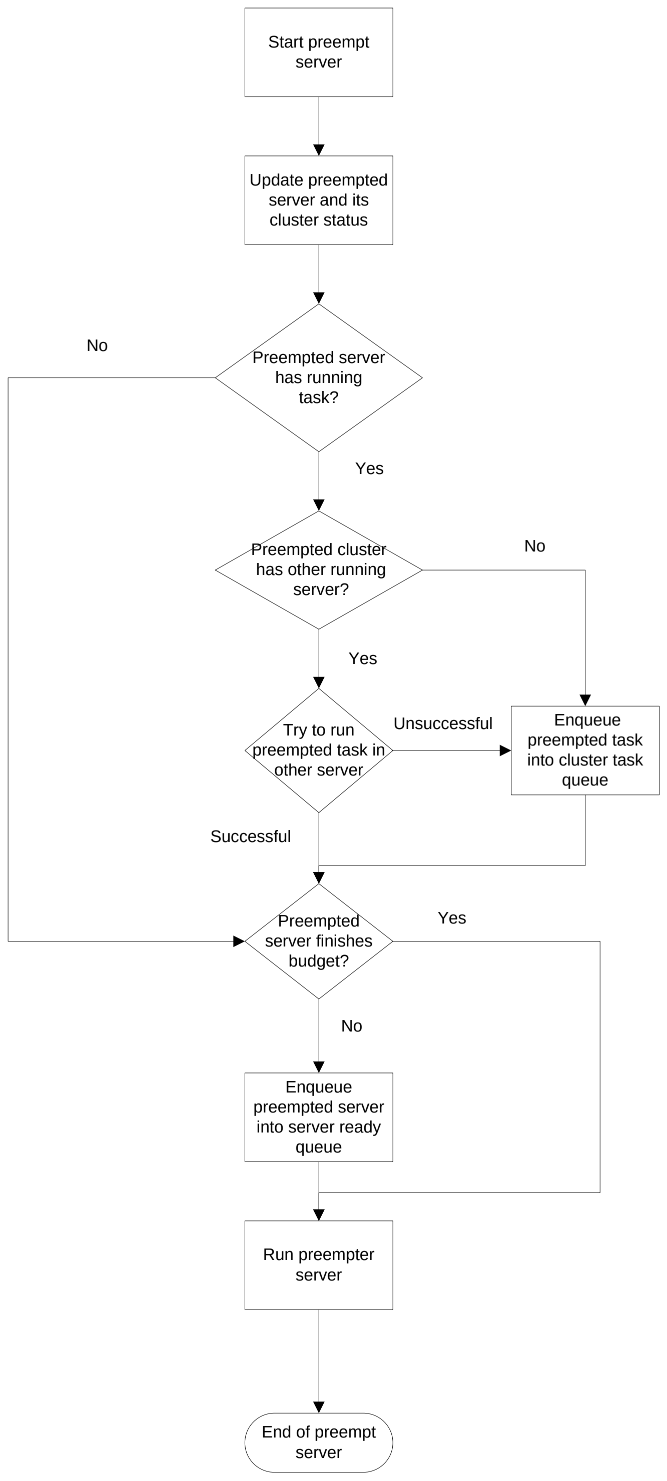
Start preempt server
Update preempted server and its cluster status
Preempted server has running task?
No
Yes
Preempted cluster has other running server?
No
Yes
Try to run preempted task in other server
Unsuccessful
Enqueue preempted task into cluster task queue
Successful
Preempted server finishes budget?
Yes
No
Enqueue preempted server into server ready queue
Run preempter server
End of preempt server

Figure 4.14: Try to run server operations

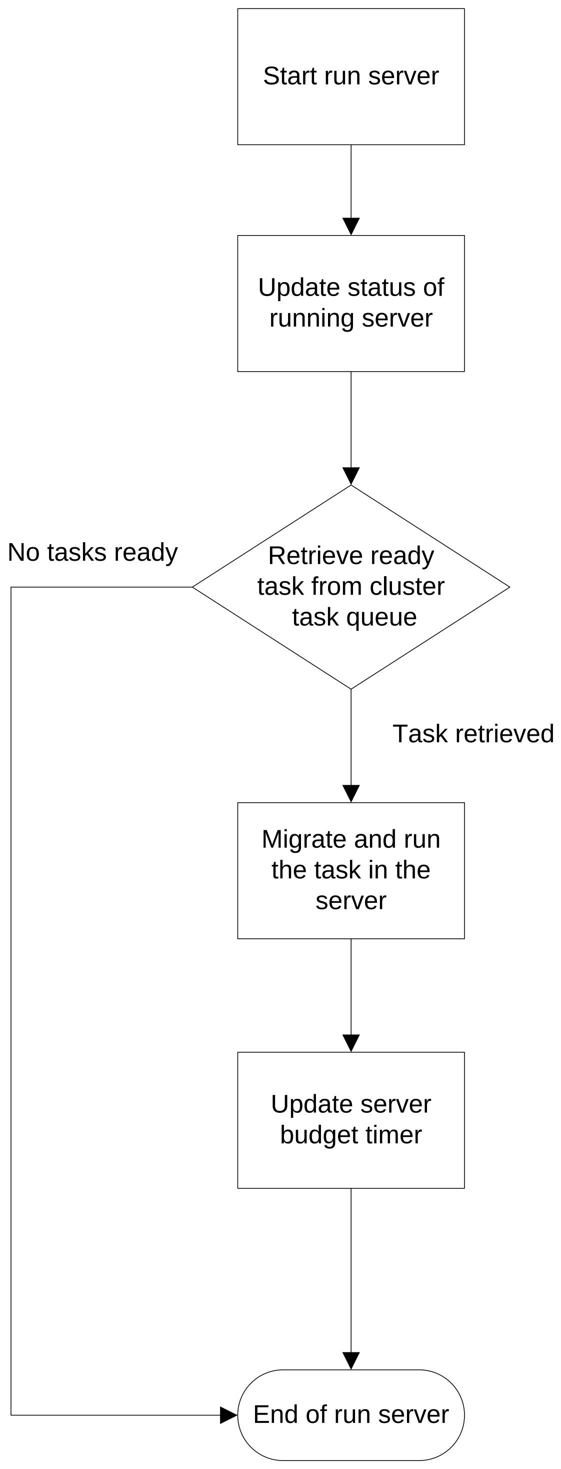
Start run server
Update status of running server
Retrieve ready task from cluster task queue
No tasks ready
Task retrieved
Migrate and run the task in the server
Update server budget timer
End of run server

Figure 4.15: Run server operations

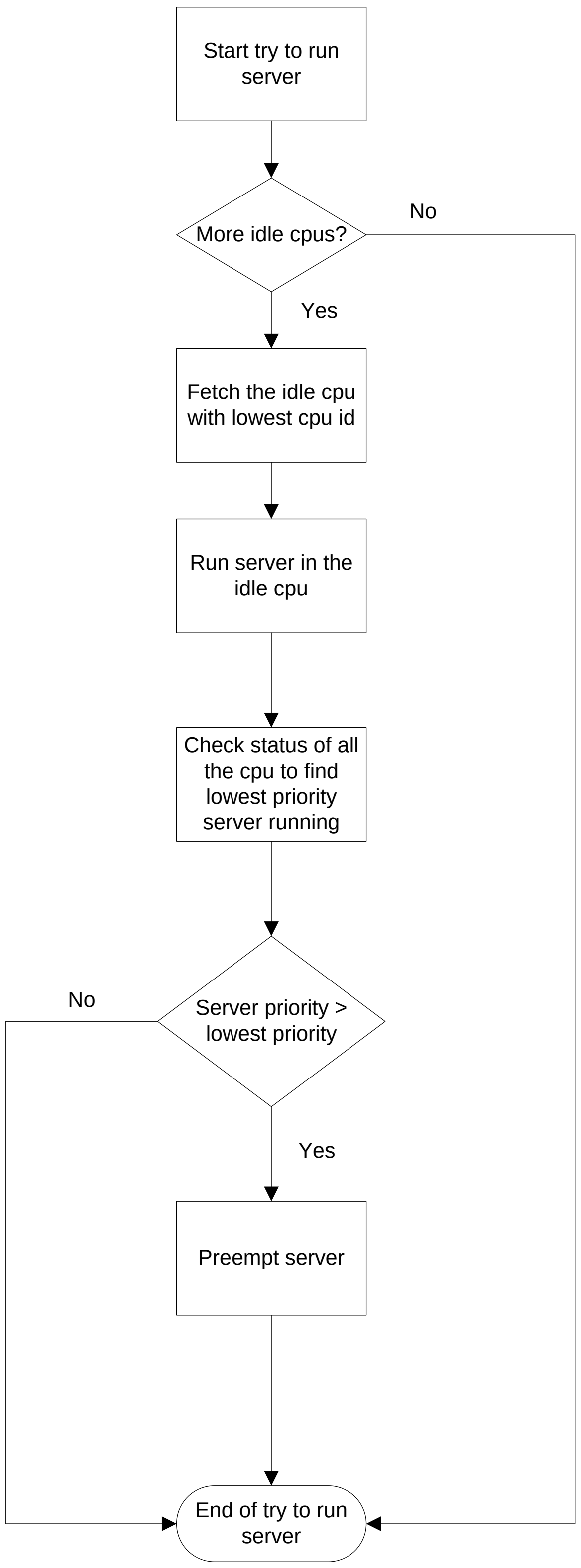
Start try to run server
More idle cpus?
No
Yes
Fetch the idle cpu with lowest cpu id
Run server in the idle cpu
Check status of all the cpu to find lowest priority server running
Server priority > lowest priority
No
Yes
Preempt server
End of try to run server

Figure 4.16: Preempt server operations

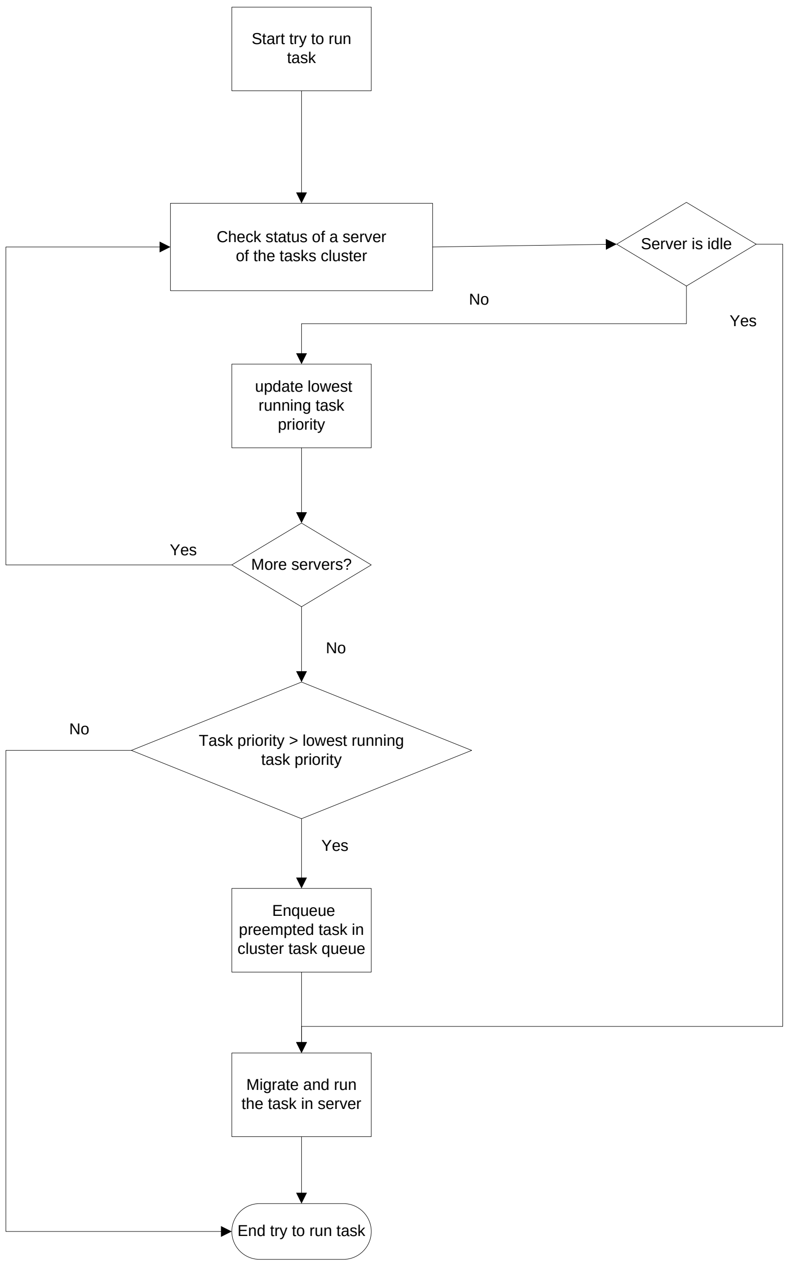
Start try to run task
Check status of a server of the tasks cluster
Server is idle
No
Yes
update lowest running task priority
Yes
More servers?
No
No
Task priority > lowest running task priority
Yes
Enqueue preempted task in cluster task queue
Migrate and run the task in server
End try to run task

Figure 4.17: Try to run task operations

Chapter 5

Results

In this chapter we present results of the experiments using our implementation of VC-HSF.

5.1 Experimental Setup

We ran our code in a Intel Core2 Duo E6700 (2.66 GHz) machine which has 2 cpus. Our base kernel version is Linux 2.6.32 recompiled with `CONFIG_PREMEPT` enabled and `jiffies` set to 1 ms. The kernel was tested using Ubuntu 10.04.

Using this setup, we have tested our code for configurations like 1-4 clusters, 1-8 servers and upto 12 real-time tasks. Various configurations of cluster periods and budgets are used to observe the effect of scheduling events like server preemption, task and server migration, etc. In all of our experiments clusters are synchronized at the beginning, which means for the first run they are released together. Real-time tasks are released before the first cluster release in some tests, in other tests tasks are configured to release after the cluster is released.

5.2 Sample Run

Now we will present a sample run of our VC-HSF. This run is done for 300 `jiffies` (ms), by 4 real-times tasks in 4 servers of 2 clusters each sharing 2 cpus. Cluster 0 with period 12 and budget 10 has two servers with budget 6 and 4. On the other hand, cluster 1 with period 32 and budget 20 has servers with budget 11 and 9. We assigned cluster 0 higher priority than the cluster 1. Then we assign three real-time tasks rttask1 (36, 1, 36), rttask2 (40, 1, 40), rttask3 (36, 1, 36) to cluster 0 and rttask4 (36, 1, 36) to cluster 1. We assigned task priorities in cluster 0 as follows: priority (rttaks1) > priority (rttaks2) > priority (rttaks3). Using `printk` function all scheduling events are printed in *dmesg*. Printed timestamp values are in `jiffies`, from which we construct the Figure 5.1. As `jiffies` values are too large we only presented the last three digits as time in the figure.

Some of the interesting events in the sample run are noted in the Figure 5.1. At $e1$, a server is migrated from one cpu to another. At $e2$, a task gets preeempted due to budget expiration of its server. Tasks also migrates to a different processor in time $e3$ and $e4$.

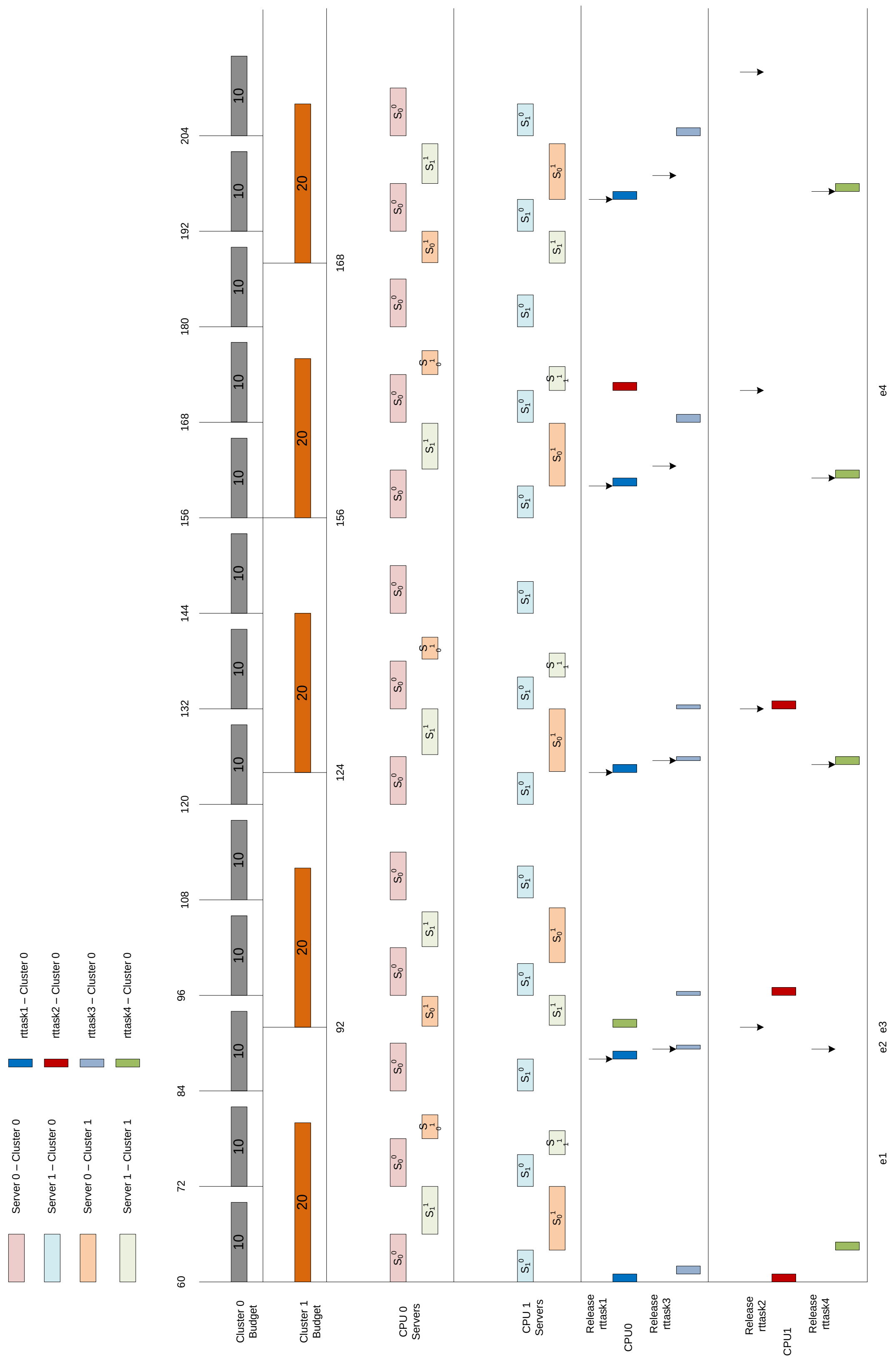

Figure 5.1: Sample run

5.3 Overhead Measurement

To measure the overhead of our VC-HSF scheduler we used coarse-grained overhead measurement. All of the functions that are not interrupt handlers are called and used from the four interrupt handlers of the scheduler (job release, job complete, cluster release, server complete). For this reason, we only measure execution time of these handlers, by timestamping the beginning and the end of their code. For timestamping we used the Linux function `getnstimeofday` which returns current time of the system with nano second level precision. We call this function just after taking global spinlock in the interrupt handler, and again call it before releasing that lock. The execution time of handler is then added to our `total_overhead` global variable. As a result, our measured total scheduler overhead in the whole run = overhead in job release handler + overhead in job complete handler + overhead in cluster release handler + overhead in server complete handler.

We ran two configurations of clusters with increasing number of tasks. These are:

- Configuration 1: In it, we used two clusters each having 2 processors. The highest priority cluster 0 had period 16 and budget 10, while the lowest priority cluster had period 32 and budget 20. As the cluster periods are harmonic, this configuration will cause very few cluster preemptions.
- Configuration 2: This configuration also used two clusters each having 2 processors. However, the highest priority cluster 0 had period 12 and budget 10, while the lowest priority cluster had period 32 and budget 20. This is different from the first configuration as the non-harmonic periods of the clusters cause a number of cluster level preemptions. This can be observed in the Figure 5.1, as we used the same cluster configuration in the sample run.

We ran both of these configurations with different number of tasks for 300 jiffies or 300 miliseconds and recorded total overhead as presented in the Table 5.1.

Table 5.1: Overhead Measurement in VC-HSF

Nr of tasks	Overhead (micro secs) configuration 1	Overhead (micro secs) configuration 2
2	1397.688	1931.848
3	1521.282	2132.633
4	1611.385	2200.346
5	1707.786	2210.893
6	1786.936	2337.634
7	1891.206	2421.915
8	2034.977	2547.566
9	2059.054	2606.388
10	2193.275	2696.545

From this coarse-grained overhead measurement, we have three simple observations:

- Preemption in the cluster level, which means the server of one cluster preempts the server of another cluster can be costly in terms of overhead. This is evident from the data we have, where only major difference between the configurations is that the second one will have more cluster level preemptions than the first one. Preemption of a cluster, can cause frequent migration of servers and tasks which may not be desirable for real-life applications.
- We used coarse-grained locking to protect the shared data structures. As shared data structures are accessed by the interrupt handlers in a fully preemptible manner, waiting for locks can be long if it is used by a complex interrupt handler.
- Increasing number of tasks does not effect scheduler overhead heavily. However, frequent task migration may effect hidden undesirable situation like "Cache thrashing".

Chapter 6

Conclusion

6.1 Summary

In this thesis we have presented design and implementation of a Virtual-Clustered Hierarchical Scheduling Framework (VC-HSF) in Linux.

We concluded that, it is possible to implement a Virtual-Clustered real-time multiprocessor scheduling algorithm in Linux without modifying the base kernel. Currently, we found no other real-time multiprocessor scheduler implementation that completely resembles scheduling operations of our scheduler. We consider our implementation not optimal in this stage as we have taken many design decisions to simplify our work. However, we know the areas where we could improve our work which will be useful for any future extension.

Our implementation can be useful in verifying extensive theoretical research done in compositional hierarchical multiprocessor scheduling [25] [35] [41] [50]. Potential application of this research is the virtualization of processor resource in real-time embedded system. We hope that our work will be helpful as a proof of concept in ongoing theoretical research in this area and will help researchers to improve their findings.

6.2 Future Work

As this is the first attempt to implement any Virtual-Clustered multiprocessor algorithm, we took many design decisions to simplify our implementation. We present some of the alternatives for these decisions as future work here:

- Although we have used ExSched scheduling framework in this implementation, very few of its functionalities are used. So, we believe a completely independent module based scheduler implementation for VC-HSF will be more flexible and efficient than our current implementation.

- We used simple task migration mechanism which requires enabling `CONFIG_PREEMPT` option. However, we can investigate more efficient task migration mechanism utilizing default Linux task migrations which may require kernel modification.

- In our code, we have used coarse-grained locking in protecting our interrupt handler to achieve synchronization. However, fine-grained locking can be used to protect only statements that access shared resources. We expect this can reduce overhead of our implementation.

- We measured coarse-grained overhead of our system. However, fine-grained overhead can be measured which includes overheads of context switch, task migration and cache related costs. We left extensive testing of our implementation to larger multiprocessor platform with more than 2 processors as future work of this thesis.
- We can use more complex data structures like heap or binary trees in the parts of our implementation where we required searching. However, this complex optimizations are left for future extension.
- Finally, we can extend our implementation of Fixed Priority Scheduling to any other Dynamic Priority scheduling algorithm for both inter-cluster and intra-cluster scheduling.

References

[1] Cgroups. Documentation/cgroups/cgroups.txt.

[2] Linux kernel documentation. https://www.kernel.org/doc/Documentation/.

[3] Real-time group scheduling. Documentation/scheduler/sched-rt-group.txt.

[4] J. H. Anderson and A. Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. In *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pages 76–85.

[5] J. H. Anderson and A. Srinivasan. Pfair scheduling: beyond periodic task systems. In *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*, pages 297–306.

[6] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, pages 193–202, March.

[7] B. Andersson and E. Tovar. Multiprocessor scheduling with few preemptions. *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on*, pages 322–334, 2006.

[8] M. Asberg, T. Nolte, and S. Kato. Towards hierarchical scheduling in linux/multi-core platform. In *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*, pages 1–4, 2010.

[9] M. Asberg, T. Nolte, S. Kato, and R. Rajkumar. Exsched: An external cpu scheduler framework for real-time systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2012 IEEE 18th International Conference on*, pages 240–249, 2012.

[10] T. P. Baker. What to make of multicore processors for reliable real-time systems? In *Proceedings of the 15th Ada-Europe international conference on Reliable Software Technologies*, Ada-Europe'10, pages 1–18. Springer-Verlag, 2010.

[11] T. P. Baker and S. K. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of Realtime and Embedded Systems*. CRC Press, 2007.

[12] S. Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *Computers, IEEE Transactions on*, 53(6):781–784, 2004.

[13] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: a notion of fairness in resource allocation. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, pages 345–354, New York, NY, USA, 1993. ACM.

[14] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, RTSS '10, pages 14–24, Washington, DC, USA, 2010. IEEE Computer Society.

[15] K. Bletsas and B. Andersson. Notional processors: An approach for multiprocessor scheduling. *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pages 3–12.

[16] J. M. Calandrino, J. H. Anderson, and D. P. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. *Real-Time Systems, 2007. ECRTS '07. 19th Euromicro Conference on*, pages 247–258, April.

[17] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. Litmus rt: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, RTSS '06, pages 111–126, Washington, DC, USA, 2006. IEEE Computer Society.

[18] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods and Models*. Chapman Hall/CRC, Boca, 2004.

[19] F. Checconi, T. Cucinotta, D Faggioli, and G. Lipari. Hierarchical multiprocessor cpu reservations for linux kernel. In *5th OSPERT Workshop*, July 2009.

[20] H. Cho, B. Ravindran, and E. D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, RTSS '06, pages 101–110, Washington, DC, USA, 2006. IEEE Computer Society.

[21] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *26th IEEE Real-Time Systems Symposium (RTSS'05)*, pages 10–398, December 2005.

[22] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):1–44, 2011.

[23] Z. Deng and J. W. S. Liu. Scheduling real-time applications in an open environment. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 308–319, Dec.

[24] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):pp. 127–140, 1978.

[25] A. Easwaran, I. Shin, and I. Lee. Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Syst.*, 43(1):25–59, 2009.

[26] D. Faggioli, M. Trimarchi, F. Checconi, and C. Scordino. An edf scheduling class for the linux kernel. In *Proceedings of the 11th Real-Time Workshop (RTLW)*, October 2009.

[27] K. Funaoka, S. Kato, and N. Yamasaki. Work-conserving optimal real-time scheduling on multiprocessors. *Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on*, pages 13–22, February.

[28] S. Funk. LRE-TL: an optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines. *Real-Time Syst.*, 46(3):332–359, 2010.

[29] S. Funk, G. Levin, C. Sadowski, I. Pye, and S. Brandt. DP-Fair: a unifying theory for optimal hard real-time multiprocessor scheduling. *Real-Time Syst.*, 47(5):389–429, 2011.

[30] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[31] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Syst.*, 25(2-3):187–205, 2003.

[32] P. Holman and J. H. Anderson. Adapting pfair scheduling for symmetric multiprocessors. *J. Embedded Comput.*, 1(4):543–564, 2005.

[33] S. Kato and N. Yamasaki. Portioned EDF-based scheduling on multiprocessors. In *Proceedings of the 8th ACM international conference on Embedded software*, pages 139–148, Atlanta, GA, USA, 2008. ACM.

[34] S. Kato and N. Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. In *Proceedings of the 2009 15th IEEE Symposium on Real-Time and Embedded Technology and Applications*, RTAS '09, pages 23–32. IEEE Computer Society, 2009.

[35] N. M. Khalilzad, M. Behnam, and T. Nolte. Exact and approximate supply bound function for multiprocessor periodic resource model: Unsynchronized servers. In *5th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'12)*, pages 1–8, December 2012.

[36] K. Lakshmanan, R. Rajkumar, and J. P. Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, pages 239–248, January.

[37] J. Lee, S. Xi, S. Chen, L. T. X. Phan, C. Gill, I. Lee, C. Lu, and O. Sokolsky. Realizing compositional scheduling through virtualization. In *Proceedings of the 18th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'12)*, pages 13–22, April 2012.

[38] S. K. Lee. On-line multiprocessor scheduling algorithms for real-time tasks. In *TENCON '94. IEEE Region 10's Ninth Annual International Conference. Theme: Frontiers of Computer Technology. Proceedings of 1994*, pages 607–611 vol.2, Aug.

[39] J. Lelli, G. Lipari, D. Faggioli, and T. Cucinotta. An efficient and scalable implementation of global edf in linux. In *7th OSPERT Workshop*, July 2011.

[40] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt. DP-FAIR: a simple model for understanding optimal multiprocessor scheduling. In *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, pages 3–13. IEEE, 2010.

[41] G. Lipari and E. Bini. A framework for hierarchical scheduling on multiprocessors: From application requirements to run-time allocation. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 249–258, 30 2010-Dec. 3.

[42] C. L. Liu. Scheduling Algorithms for Multiprocessors in a Hard Real-Time Environment. *JPL Space Programs Summary 37-60*, II:28–31, 1969.

[43] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[44] J. M. Lopez, J. L. Diaz, and D. F. Garcia. Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Real-Time Syst.*, 28(1):39–68, 2004.

[45] R. McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6(1):1–12, 1959.

[46] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Cambridge, MA, USA, 1983.

[47] A. K. Mok, X. Feng, and C. Deji. Resource partition for real-time systems. In *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS '01)*, pages 75–84, May 2001.

[48] C. A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation (extended abstract). In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 140–149. ACM, 1997.

[49] X. Qi, D. Zhu, and H. Aydin. Cluster scheduling for real-time systems: utilization bounds and run-time overhead. *Real-Time Systems*, 47:253–284, 2011.

[50] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, pages 181–190. IEEE Computer Society, 2008.

[51] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the 24th IEEE Real-Time Systems Symposium, (RTSS '03)*, pages 2 – 13, 2003.

[52] A. Srinivasan and S. Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Inf. Process. Lett.*, 84(2):93–98, 2002.

[53] J. Yang, H. Kim, S. Park, C. Hong, and I. Shin. Implementation of compositional scheduling framework on virtualization. *SIGBED Rev.*, 8(1):30–37, March 2011.

[54] D. Zhu, D. Mosse, and R. Melhem. Multiple-resource periodic scheduling problem: how much fairness is necessary? In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 142–151, Dec.

[55] H. Zhu, S. Goddard, and M. B. Dwyer. Response time analysis of hierarchical scheduling: The synchronized deferrable servers approach. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 239–248, 2011.

Appendix A

VC-HSF

A.1 Guidelines for running VC-HSF

1. Compile your Linux Kernel with `CONFIG_PREEMPT` option enabled and `jiffies` set to 1 ms.
2. Follow README instructions of ExSched and compile its core module.
3. Go to cluster folder of the plugin section.
4. Configure real-time tasks that you need to run in start.sh of the tasks folder.
5. Configure clusters that you want to run in vchsf.c file.
6. Execute using vchsf-start.sh file.
7. You can find output of the run via dmesg command.

A.2 Source code listing of VC-HSF

The implementation of VC-HSF includes following major source files:

1. `vchsf.c`: The main file where all the handlers and functions of the scheduler are implemented.
2. `release-queue.c`: The file where cluster release queue is defined.
3. `ready-queue.c`: In this file cluster ready queue and its functions are defined.
4. `server-queue.c`: In this file server ready queue and its functions are defined.
5. `task-queue.c`: In this file task ready queue and its functions are defined.

Other than that, core files of ExSched are also modified. Here we present the main file of our implementation `vchsf.c`.

```
#include <linux/string.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/time.h>
#include <linux/sched.h>
#include <linux/slab.h>
//#include <../../kernels/2.6.31-20-generic-
pae/include/resch/config.h>
//#include <../../kernels/2.6.31-20-generic-pae/include/resch/core.h>
//#include </usr/src/kernels/2.6.32-38-
generic/include/resch/config.h>
//#include </usr/src/kernels/2.6.32-38-generic/include/resch/core.h>
#include </usr/src/linux-2.6.32/include/resch/config.h>
#include </usr/src/linux-2.6.32/include/resch/core.h>
//#include </usr/src/linux-headers-3.5.0-17-
generic/include/resch/config.h>
//#include </usr/src/linux-headers-3.5.0-17-
generic/include/resch/core.h>
#include <../arch/x86/include/asm/div64.h>
#include <resch/config.h>
#include <resch/core.h>

MODULE_LICENSE("Dual BSD/GPL");
MODULE_DESCRIPTION("VC-HSF Scheduler");
MODULE_AUTHOR("Jakaria Abdullah");

#define WITH_TASKS

#define SYSTEM_TIMEOUT 10000    // the whole server based scheduling
start after 10 ms

/* Bits used by our hierarchical scheduler in the hsf flag in RESCH
task control block */
#define ACTIVATE 0
#define RESCH_PREVENT_RELEASE 1
#define PREV_RELEASE_SKIPPED 2
#define PREVENT_RELEASE_AT_INIT 3

#define MAX_RELEASED_AT_SAME_TIME 20    /* Maximum nr of servers that
can be released at the same time instance */
#define MAX_NR_OF_TASKS_IN_SERVER 10
#define MAX_NR_OF_TASKS_IN_CLUSTER 10
#define MAX_NR_OF_CLUSTERS 2
#define TRUE            1
#define FALSE           0

// cluster states
#define CLUSTER_ACTIVE 1
#define CLUSTER_READY   2
#define CLUSTER_EXPIRED 3
#define CLSUTER_PREEMPTED 4

// CPU States
#define CPU_BUSY 8
#define CPU_IDLE 9

// vc-hsf parameters
#define NR_OF_SERVERS 6
#define NR_OF_CLUSTERS 2
#define MAX_NR_SERVERS_CLUSTER 3
#define RUN_JIFFIES 300

/* we have ONE central timer combined with our own timer-queue, i.e.,
release queue. */
struct timer_list event_timer; // This timer is used for invoking a
server release and also budget expirations
```

```
typedef struct vc_server_struct server_t;
typedef struct vc_cluster_struct cluster_t;
typedef struct vc_task_struct vctask_t;
typedef struct cpu_resource_struct cpu_t;

struct vc_task_struct {
      resch_task_t *rt;
      int priority;
      vctask_t *next;
};

#include "task-queues.c"  // Implementation of task queues

vctask_t vc_task[NR_RT_TASKS];

struct vc_server_struct {
        int id; // 0 to (INT_SIZE_BIT-1)
        cluster_t * mycluster;  // new
        int server_cpu;   // new
        int period;
        char running; // TRUE if the server is running, otherwise
FALSE
        int budget;
        int remain_budget;
        unsigned long budget_expiration_time;
        struct timer_list server_budget_timer;
        int priority;
        unsigned long timestamp;
        vctask_t *active_task;
        server_t *next;
};

#include "server-queues.c"  // Implementation of server queues

struct vc_cluster_struct {
      int vc_cluster_id;
      int cluster_state;  // running flag to indicate cluster is
running
      int nr_cpus;  // m
      int period;       // period
      int budget; // theta
      int priority;
      int remain_budget;
      int nr_active_servers;
      int nr_ready_servers;
      //struct timer_list cluster_period_timer;
      server_t SERVERS[MAX_NR_SERVERS_CLUSTER];
        tpq READY_TASKS;
      //spinlock_t lock;   // lock used to access server
      //vctask_t *resch_task_list[MAX_NR_OF_TASKS_IN_CLUSTER];
      //resch_task_t *resch_task_list[MAX_NR_OF_TASKS_IN_CLUSTER]; //
Maximum nr of tasks per cluster...
      cluster_t *next;
};

//static tpq READY_TASKS[NR_OF_CLUSTERS];

#include "ready-queue.c" // Implementation of cluster ready queue

#include "release-queue.c" // Implementation of cluster release queue


//server_t SERVERS[NR_OF_SERVERS];  // Here are the servers, they
will also reside in the server ready queue
cluster_t CLUSTERS[NR_OF_CLUSTERS];
```

```
struct cpu_resource_struct{
      int cpu_busy;                //  status to show whether any
server of virtual cluster is running there
      int active_cluster_priority;
      server_t * active_server;
};


cpu_t cpu_status[NR_RT_CPUS];    // running  status of cpus, 0 =
idle, 1 = busy
static int idle_cpus;               //  global variable for idle cpu
count
static int lowest_active_cluster_prio;  // global variable for lowest
active cluster priority
static int cluster_complete_on_preemption;
unsigned long total_overhead;
//cluster_t ACTIVE_CLUSTERS[NR_OF_CLUSTERS];   // global list of
active servers
spinlock_t global_lock;


// Main structure of the release queue
static relPq CLUSTER_RELEASE_QUEUE;
// Nodes residing in the release queue which represent a specific
server
relNode RelNodes[NR_OF_CLUSTERS];

static pq CLUSTER_READY_QUEUE; // This is the instance of the server
ready queue
static spq SERVER_READY_QUEUE;


// list of function declarations
void task_run(resch_task_t *rt);
void job_release(resch_task_t *rt);
void job_complete(resch_task_t *rt);
void init_cluster(cluster_t *cluster, int c_id, int nr_cpus, int
budget_msecs, int period_msecs, int priority);
int check_idle_cpus(void);
void init_server(server_t * server, cluster_t *cluster, int id);
server_t *check_active_clusters_for_servers(void);
void server_complete_handler(unsigned long __data);
int try_to_run_task(vctask_t *preemptee_task, cluster_t *cluster);
int run_server(server_t *server, int cpu);
int preempt_server(server_t *server, int cpu);
int try_to_run_server(server_t *server);
void stop_server(server_t *server);
void update_lowest_cluster_prio(void);
void assign_server_budget(cluster_t *cluster, int nr_cpus);
void cluster_release_handler(unsigned long __data);
void cluster_complete_handler(cluster_t* complete);
void migrate_vchsf_task(resch_task_t *rt, int cpu_dst);

// function for inserting timer

/*

inline void insert_timer(struct timer_list *timer, void (*my_handler)
(unsigned long),  unsigned long data, unsigned long expire_time)
{
  int timer_status = timer_pending(timer);
  struct Handler_Data *data_handler;
  data_handler = (struct Handler_Data *) data;
  data_handler->timestamp = expire_time;
  //print_int("insert_timer","timer status", timer_status);
  if(&timer->entry == NULL)
  {
      //print_warning("insert_timer","timer is NULL");
      //scheduling_error = -1;
      return;
  }
  if(expire_time < jiffies)
  {
      //print_warning("insert_timer","expire_time < jiffies");
      //scheduling_error = -1;
      return;
  }
  //print_long("insert_timer","setting timer to", expire_time);
  if(timer_status == 1 && timer->expires == jiffies)
  {
      // Pending
      //print_warning("insert_timer","setting timer which expires
now");
      timer->data = data;
      mod_timer(timer, expire_time);
  }
  else
  {
      setup_timer_on_stack(timer, my_handler, data);
      mod_timer(timer, expire_time);
  }
}

*/

// function for removing timer

inline void remove_timer(struct timer_list *timer)
{
//   if(timer->expires == jiffies)
//   {
//      print_warning("remove_timer", "Error: timer expires now");
//      scheduling_error = -1;
//      return;
//   }
    del_timer(timer);
    //del_timer_sync(timer);
}



/*
 *
 * plugin function 1
 *
 * this is called by api_run
 * 1. initialize vctask_t
 * 2. #insert vctask_t into ready queue of the cluster -NO
 * 3. other flags need to be handled
 *
 */


void task_run(resch_task_t *rt) {

      //migrate_task(rt, 0); // Migrate all tasks to CPU:0 for now...
        unsigned long flags;
      printk(KERN_WARNING "task_create: %d %s %d %d period:%lu wcet:%
lu (%lu)\n", rt->pid, rt->task->comm, rt->prio, rt->cpu_id, rt->
period, rt->wcet, jiffies);
        printk(KERN_WARNING "task_rid: %d (%lu)\n", rt->rid,
jiffies);

      //printk(KERN_INFO "HSF: TASK_RUN!\n");
        spin_lock_irqsave(&global_lock, flags);
        //preempt_disable();
      vc_task[rt->rid].rt = rt;
      vc_task[rt->rid].priority = rt->prio - 68;

      //insert task into ready queue of its cluster - NOT NOW
      //bitmap_insert_tpq(&CLUSTERS[rt->cluster_id].READY_TASKS,
vc_task[rt->rid]);

        //preempt_enable();
        spin_unlock_irqrestore(&global_lock, flags);
}


/*
 * plugin function 2: job release
 * 1. if the job is higher priority than the current running job its
cluster preempt
 * 2. otherwise insert the job into ready queue of its cluster
 */
void job_release(resch_task_t *rt) {

      //server_t *highest_prio_server;
      unsigned long flags;
            int FLAG = FALSE;
        struct timespec tmp_time_start, tmp_time_end, overhead;

        //printk(KERN_WARNING "Inside job_release (%lu)!!!!",
jiffies);

      spin_lock_irqsave(&global_lock, flags);
        getnstimeofday(&tmp_time_start);
        //preempt_disable();

        if(rt == NULL){
            printk(KERN_WARNING "VC-HSF: job released with empty
rt!!!!");
            //preempt_enable();
            spin_unlock_irqrestore(&global_lock, flags);
            return;
        }

      if(CLUSTERS[rt->cluster_id].cluster_state == CLUSTER_ACTIVE){
            // means cluster of this task is activated, so may be
server is running
            if(CLUSTERS[rt->cluster_id].nr_active_servers !=0){
                    if((try_to_run_task(&vc_task[rt->rid], &CLUSTERS
[rt->cluster_id]))== FALSE){
                                    FLAG = TRUE;
                      }else{
                              //vchsf_job_start(rt);
                              printk(KERN_WARNING "job running in
release: %d %s %d cpu:%d server:%d cluster:%d (%lu)\n", rt->pid, rt->
task->comm, rt->prio, rt->cpu_id, rt->server_id, rt->cluster_id,
jiffies);
                          }
            }
      }else{
          printk(KERN_WARNING "job released but cluster not active so
sleep: %d %d (%lu)\n", rt->pid, rt->cpu_id, jiffies);
          FLAG = TRUE;
        }
```

```
    if(FLAG == TRUE){   // job need to be queued into ready queue of
its cluster
          //dequeue_task(running_task->rt);
         #ifdef WITH_TASKS
       rt->task->state = TASK_UNINTERRUPTIBLE;
            //set_tsk_need_resched(rt->task);
         #endif
      bitmap_insert_tpq(&CLUSTERS[rt->cluster_id].READY_TASKS,
&vc_task[rt->rid]);

        printk(KERN_WARNING "job queued in release: %d %s %d %d %d (%
lu)\n", rt->pid, rt->task->comm, rt->prio, rt->cpu_id, rt->server_id,
jiffies);
    }

    //preempt_enable();
    getnstimeofday(&tmp_time_end);
    overhead = timespec_sub(tmp_time_end, tmp_time_start);
    total_overhead += overhead.tv_nsec + overhead.tv_sec*1000000000;
    spin_unlock_irqrestore(&global_lock, flags);

}


void migrate_vchsf_task(resch_task_t *rt, int cpu_dst){
        rt->cpu_id =cpu_dst;
      /* task migration occurs here. */
      cpus_clear(rt->cpumask);
      cpu_set(cpu_dst, rt->cpumask);
      set_cpus_allowed_ptr(rt->task, &rt->cpumask);
}

/*
 * plugin function 3: job complete
 *
 * 1. make task dequeue from the processor
 * 2. dequeue task from cluster ready queue and try to run in the
server cpu
 * 3.
 *
 *
 */

void job_complete(resch_task_t *rt) {

      // needs spinlock

      vctask_t *running_task;
      //vctask_t *complete_task;
      resch_task_t *new_rt;
        int current_cpu;
      unsigned long flags;
        struct timespec tmp_time_start, tmp_time_end, overhead;
        server_t * current_server = NULL;

        //printk(KERN_WARNING "Inside job_complete (%lu)!!!!",
jiffies);

      spin_lock_irqsave(&global_lock, flags);
        getnstimeofday(&tmp_time_start);
        //preempt_disable();

        if(rt == NULL){
            printk(KERN_WARNING "VC-HSF: job complete with empty
rt!!!!");
            //preempt_enable();
            spin_unlock_irqrestore(&global_lock, flags);
            return;
        }


      current_cpu = rt->cpu_id;
      current_server = cpu_status[current_cpu].active_server;

    // need to dequeue the completed task and put into sleep
     #ifdef WITH_TASKS
        //dequeue_task(rt);
        rt->task->state = TASK_UNINTERRUPTIBLE;
      set_tsk_need_resched(rt->task);
     #endif
        printk(KERN_WARNING "job complete: %d %s %d %d %d (%lu)\n",
rt->pid, rt->task->comm, rt->prio, rt->cpu_id, rt->server_id,
jiffies);

      if((running_task = bitmap_retrieve_tpq(&CLUSTERS[rt->
cluster_id].READY_TASKS))==NULL){
            // means there is no more ready task jobs in cluster task
queue so remain idle
            current_server->active_task = NULL;
      }else{

            current_server->active_task = running_task;
            new_rt = running_task->rt;
                new_rt->server_id = current_server->id;
                #ifdef WITH_TASKS
                migrate_vchsf_task(new_rt, current_cpu);
                if(new_rt->task->state != TASK_RUNNING){
                    wake_up_process(new_rt->task);
                }
            //migrate_task(resch_task_t *rt, int cpu_dst)
                //enqueue_task(new_rt);
            //migrate_task(new_rt, current_cpu);
                //vchsf_job_start(rt);
                #endif
                printk(KERN_WARNING "job running in complete: %d %s %
d cpu:%d server:%d cluster:%d (%lu)\n", new_rt->pid, new_rt->task->
comm, new_rt->prio, new_rt->cpu_id, new_rt->server_id, new_rt->
cluster_id, jiffies);
      }
        //preempt_enable();
        getnstimeofday(&tmp_time_end);
        overhead = timespec_sub(tmp_time_end, tmp_time_start);
        total_overhead += overhead.tv_nsec + overhead.tv_sec*
1000000000;
        spin_unlock_irqrestore(&global_lock, flags);
}



/*
 * 1. find the active server running  lowest priority task
 * 2. if priority of that task is lower than tasks priority than
preempt that task
 */

int try_to_run_task(vctask_t *preemptee_task, cluster_t *cluster){

      int i;
      int min_priority, index;
      vctask_t * preempted_task;
      //resch_task_t * rt;
      int cpu_min = 0;
      index = 0;

      //min_priority = preemptee_task->rt->prio;
        min_priority = preemptee_task->priority;  // changed newly
      for(i = 0; i < cluster->nr_cpus; i++){
            if(cluster->SERVERS[i].running == TRUE){
                   if(cluster->SERVERS[i].active_task == NULL){
                         cluster->SERVERS[i].active_task =
preemptee_task;
                         #ifdef WITH_TASKS
                         preemptee_task->rt->server_id = i;
                         migrate_vchsf_task(preemptee_task->rt,
cluster->SERVERS[i].server_cpu);
                         if(preemptee_task->rt->task->state !=
TASK_RUNNING){
                              wake_up_process(preemptee_task->rt->
task);
                         }

                        //enqueue_task(preemptee_task->rt);
                        //migrate_task(preemptee_task->rt,cluster->
SERVERS[i].server_cpu);
                        #endif
                        return TRUE;
                    }else{
                        if(cluster->SERVERS[i].active_task->priority
> min_priority){
                        min_priority = cluster->SERVERS
[i].active_task->priority;
                        cpu_min = cluster->SERVERS[i].server_cpu;
                        index = i;
                  }
                    }
            }
      }

      if(min_priority > preemptee_task->priority){
            // means our task can preempt this task
            preempted_task = cluster->SERVERS[index].active_task;
            #ifdef WITH_TASKS
          //dequeue_task(preempted_task->rt);
              preempted_task->rt->task->state = TASK_UNINTERRUPTIBLE;
            set_tsk_need_resched(preempted_task->rt->task);
          #endif
              bitmap_insert_tpq(&cluster->READY_TASKS,
preempted_task);
                printk(KERN_WARNING "job preempted: %d %s %d %d %d
\n", preempted_task->rt->pid, preempted_task->rt->task->comm,
preempted_task->rt->prio,preempted_task->rt->cpu_id, preempted_task->
rt->server_id);
            cluster->SERVERS[index].active_task = preemptee_task;
            //migrate_task(resch_task_t *rt, int cpu_dst)
                #ifdef WITH_TASKS
                preemptee_task->rt->server_id = index;
                migrate_vchsf_task(preemptee_task->rt, cpu_min);
                if(preemptee_task->rt->task->state != TASK_RUNNING){
                wake_up_process(preemptee_task->rt->task);
                }

                //enqueue_task(preemptee_task->rt);
                //migrate_task(preemptee_task->rt,cpu_min);
                #endif
              printk(KERN_WARNING "job running in try to preempt: %d
%s %d cpu:%d server:%d cluster:%d (%lu)\n", preemptee_task->rt->pid,
preemptee_task->rt->task->comm, preemptee_task->rt->prio,
preemptee_task->rt->cpu_id, preemptee_task->rt->server_id,
preemptee_task->rt->cluster_id, jiffies);
            return TRUE;
      }

      return FALSE;
}


/*
 * procedure which initialises a cluster data structure
 *
 */


void init_cluster(cluster_t *cluster, int c_id, int nr_cpus, int
budget_msecs, int period_msecs, int priority) {

      int i;
      cluster->vc_cluster_id = c_id;
      cluster->cluster_state = CLUSTER_READY;
      cluster->nr_cpus = nr_cpus;
      cluster->period = msecs_to_jiffies(period_msecs);
      cluster->budget = msecs_to_jiffies(budget_msecs);
      cluster->remain_budget = cluster->budget;
      cluster->nr_active_servers = 0;
      cluster->nr_ready_servers = 0;
      cluster->priority = priority;
      //cluster->resch_task_list = NULL;
      for(i=0; i < cluster->nr_cpus; i++){
            init_server(&cluster->SERVERS[i],cluster, i);   // now
stub
      }

      init_tpq(&cluster->READY_TASKS);
      printk(KERN_WARNING "VC-HSF: Cluster(%d) initialized with
budget %d remaininig budget %d (%lu) \n", cluster->vc_cluster_id,
cluster->budget, cluster->remain_budget, jiffies);
}


int check_idle_cpus(void){
      int i;
      for(i = 0; i < NR_RT_CPUS; i++){
            if(cpu_status[i].cpu_busy == CPU_IDLE){
                  return i;
            }
      }
      return FALSE;   // this should be impossible cpu value that can
be returned
}


// stub
void init_server(server_t * server, cluster_t *cluster, int id){

    server->id = id;   // id respective to cluster
    server->mycluster = cluster;
    server->server_cpu = 0;   // by default initialized to 0
    server->period = cluster->period;
    server->running = FALSE; // TRUE if the server is running,
otherwise FALSE
    server->budget = 0;  // default not assigned a budget
    server->remain_budget = 0;   // default no remaining budget
    server->budget_expiration_time = 0;
    init_timer(&server->server_budget_timer);
    server->priority = cluster->priority;
    server->timestamp = 0;   // this should be used with jiffies in
run_server
    server->active_task = NULL;  // no active task at init
    printk(KERN_WARNING "VC-HSF: Server(%d) of cluster (%d)
initialized (%lu)\n", server->id,server->mycluster->vc_cluster_id,
jiffies);
}

// stub may be inline function can be implemented
server_t *check_active_clusters_for_servers(void){

      server_t *server;
      server = bitmap_retrieve_spq(&SERVER_READY_QUEUE);
      if(server != NULL){
            return server;
      }
      return NULL;
}

/*
 *
 * server complete handler is called by the budget expiration timer
 * 1. remove server from cpu
 * 2. stop running task in the server
 * 3. try to run that task in other server of the same cluster
 * 4. if cluster of the server completes than call cluster_complete
handler
 * 5. otherwise pick a new server from server ready queue and run it
on the cpu
 */

void server_complete_handler(unsigned long __data) {

      // needs spinlock


        unsigned long flags;
      server_t *completed = (server_t *)__data;
      server_t *to_run_server;
      vctask_t *running_task;
      //resch_task_t *rt;
      cluster_t * temp_cluster;
        struct timespec tmp_time_start, tmp_time_end, overhead;
      int cpu = completed->server_cpu;
      int FLAG = FALSE;
      int j;

      spin_lock_irqsave(&global_lock, flags);
        getnstimeofday(&tmp_time_start);
        //preempt_disable();

      cpu_status[cpu].cpu_busy = CPU_IDLE;
      idle_cpus++;
      cpu_status[cpu].active_cluster_priority = 0;    // this one can
be potential bug
      cpu_status[cpu].active_server = NULL;

      // make the server stopped on that processor
      completed->running = FALSE;
      remove_timer(&completed->server_budget_timer);
      //destroy_timer_on_stack(&completed->server_budget_timer);
      //server->server_cpu = cpu;

      // schedule budget timer of the server
      completed->budget_expiration_time = 0; // put remaining
      completed->timestamp = jiffies; // Set timestamp in case of
```

```
preemption (budget accounting)

      completed->mycluster->nr_active_servers--;

      if(completed->active_task != NULL){   // means task is running
in that server
           running_task = completed->active_task;
           // try to run preemted task on other servers of its
cluster
           if(completed->mycluster->nr_active_servers !=0){
               // there are other servers from this cluster running
try to run there
               if((try_to_run_task(running_task, completed->
mycluster))== FALSE){
                         FLAG = TRUE;
               }else{
                       //vchsf_job_start(running_task->rt);
                       printk(KERN_WARNING "job running in server
complete: %d %s %d cpu:%d server:%d cluster:%d (%lu)\n",
running_task->rt->pid, running_task->rt->task->comm, running_task->
rt->prio, running_task->rt->cpu_id, running_task->rt->server_id,
running_task->rt->cluster_id, jiffies);
                   }
           }else{
               FLAG = TRUE;
           }
           if(FLAG == TRUE){   // means you need to dequeue that task
and put it into cluster ready_queue
               #ifdef WITH_TASKS
                   //dequeue_task(running_task->rt);
                running_task->rt->task->state = TASK_UNINTERRUPTIBLE;
                set_tsk_need_resched(running_task->rt->task);
                #endif
                     bitmap_insert_tpq(&completed->mycluster->
READY_TASKS, running_task);
                   printk(KERN_WARNING "job queued in server
complete: %d %s %d %d %d (%lu)\n", running_task->rt->pid,
running_task->rt->task->comm, running_task->rt->prio,running_task->
rt->cpu_id, running_task->rt->server_id, jiffies);
           }
      }

      if((completed->mycluster->nr_active_servers == 0)&&(completed->
mycluster->nr_ready_servers == 0)){
            // means the cluster has no running or active server so
it is completed
            cluster_complete_handler(completed->mycluster);
      }else{
            // mean cluster is not completed so we need to pick a
ready server and run it
            // check for a active cluster which has ready servers
          while(idle_cpus != 0){
            if((to_run_server = check_active_clusters_for_servers())
== NULL){
                  break;
              }else{
                  //cpu = check_idle_cpus();
                        if((try_to_run_server(to_run_server))== TRUE)
{         // stub
                              to_run_server->running = TRUE;
                              to_run_server->mycluster->
nr_active_servers++;
                        }
              }
          }
```

```
            // this means we need to launch a ready cluster
          while(idle_cpus != 0){
            if((temp_cluster = bitmap_retrieve(&CLUSTER_READY_QUEUE))
== NULL){
                  // there is no more ready clusters in the system to
launch
                  printk(KERN_WARNING "NO Cluster in ready queue (%
lu)\n", jiffies);
                  break;
            }else{
                   temp_cluster->cluster_state = CLUSTER_ACTIVE;
                         temp_cluster->remain_budget = temp_cluster->
budget;
                   assign_server_budget(temp_cluster, temp_cluster->
nr_cpus);
                   for(j =0; j < temp_cluster->nr_cpus; j++){
                         if((try_to_run_server(&temp_cluster->SERVERS
[j]))== TRUE){
                              temp_cluster->SERVERS[j].running =
TRUE;
                              temp_cluster->nr_active_servers++;
                           }

                      if(temp_cluster->SERVERS[j].running != TRUE){
                           temp_cluster->SERVERS[j].running = FALSE;
                           bitmap_insert_spq(&SERVER_READY_QUEUE,
&temp_cluster->SERVERS[j]);
                            temp_cluster->nr_ready_servers++;
                      }
                 }
                   update_lowest_cluster_prio();  // updating lowest
cluster priority
            }
          }
          update_lowest_cluster_prio();
      }

        printk(KERN_WARNING "server_complete: server:%d cluster:%d
cpu:%d (%lu)\n",completed->id, completed->mycluster->vc_cluster_id,
completed->server_cpu, jiffies);
        //preempt_enable();
        getnstimeofday(&tmp_time_end);
        overhead = timespec_sub(tmp_time_end, tmp_time_start);
        total_overhead += overhead.tv_nsec + overhead.tv_sec*
1000000000;
        spin_unlock_irqrestore(&global_lock, flags);
}




// stub
/*
 * this function is to run a server in a idle cpu
 * 1.change the status of the processor running server
 * 2.adjust budget of the server
 * 3.retrieve ready task from cluster task queue
 * 4.migrate task to the idle cpu to run
 * 5.enable budget timer for server completion
 *
 *
 */

int run_server(server_t *server, int cpu){

    vctask_t * running_task;
```

```
      resch_task_t *rt;

        printk(KERN_WARNING "run_server idle cpus before:%d
\n",idle_cpus);
      if(cpu_status[cpu].cpu_busy != CPU_IDLE){
            return FALSE;
      }else{
            cpu_status[cpu].cpu_busy = CPU_BUSY;
            idle_cpus--;
            cpu_status[cpu].active_cluster_priority = server->
priority;
            cpu_status[cpu].active_server = server;

            // make the server running on that processor
            server->running = TRUE;
            server->server_cpu = cpu;

            // schedule budget timer of the server
            server->budget_expiration_time = jiffies+ server->
remain_budget; // put remaining
            server->timestamp = jiffies; // Set timestamp in case of
preemption (budget accounting)

            server->mycluster->nr_ready_servers--;
            server->mycluster->nr_active_servers++;
            //  here we need to fetch tasks from cluster task queue
and run that task
            if(server->mycluster->cluster_state != CLUSTER_ACTIVE){
                  server->mycluster->cluster_state = CLUSTER_ACTIVE;
            }

            if((running_task = bitmap_retrieve_tpq(&server->
mycluster->READY_TASKS))==NULL){
                  // means there is no more ready task jobs in
cluster task queue so remain idle
                  server->active_task = NULL;
            }else{
                  server->active_task = running_task;
                  rt = running_task->rt;
                  //migrate_task(resch_task_t *rt, int cpu_dst)
                        #ifdef WITH_TASKS
                        rt->server_id = server->id;
                        //enqueue_task(rt);
                  //migrate_task(rt, cpu);
                        //vchsf_job_start(rt);

                        migrate_vchsf_task(rt, cpu);
                        if(rt->task->state != TASK_RUNNING){
                           wake_up_process(rt->task);
                        }

                        #endif
                        printk(KERN_WARNING "job running in run
server: %d %s %d cpu:%d server:%d cluster:%d (%lu)\n", rt->pid, rt->
task->comm, rt->prio, rt->cpu_id, rt->server_id, rt->cluster_id,
jiffies);

            }

            //insert_timer(&server->server_budget_timer,
server_complete_handler, (unsigned long)server, server->
budget_expiration_time);
            setup_timer_on_stack(&server->server_budget_timer,
server_complete_handler, (unsigned long)server);
            mod_timer(&server->server_budget_timer, server->
budget_expiration_time);
```

```
      }
        printk(KERN_WARNING "run_server idle cpus after:%d
\n",idle_cpus);
        printk(KERN_WARNING "server starts running with budget:%d
remain budget:%d expiration time:(%lu) (%lu)\n",server->budget,
server->remain_budget, server->budget_expiration_time, jiffies);
        printk(KERN_WARNING "server starts running: server:%d
cluster:%d cpu:%d (%lu)\n",server->id, server->mycluster->
vc_cluster_id, server->server_cpu, jiffies);
      return TRUE;
}


/*
 * stub: this function is used to stop server other than budget
comepletion
 * in our case it will be used by server preemption function
 * 1. Before calling this function preempted server has already
removed
 * 2. Just need to call cluster complete
 */

void stop_server(server_t *server){

   if((server->mycluster->nr_active_servers == 0)&&(server->
mycluster->nr_ready_servers == 0)){
            // means the cluster has no running or active server so
it is completed
                cluster_complete_on_preemption = TRUE;
            cluster_complete_handler(server->mycluster);
   }
}

/*
 * stub : this function is used preempt a server which  is running on
cpu
 * 1. remove preempted server from cpu, put it into server ready
queue
 * 2. if the server is running some task, fetch it and put it into
cluster task ready queue
 * 3. adjust remaining budget of the server
 * 4. adjust remaining budget of the server's cluster : not needed
may be
 * 5. change status of the server
 * 6. if needed change status of the cluster
 * 7. adjust the lowest priority cluster flag
 *
 */

int preempt_server(server_t *server, int cpu){

      server_t * preempted_server;
      vctask_t * preempted_task;
      vctask_t * running_task;
      resch_task_t * rt;
      cluster_t * preempted_cluster;
      int FLAG = FALSE;
        //int debug = 0;

      preempted_server = cpu_status[cpu].active_server;
      preempted_cluster = preempted_server->mycluster;
      preempted_cluster->nr_active_servers--;

      remove_timer(&preempted_server->server_budget_timer);
      //destroy_timer_on_stack(&preempted_server->
server_budget_timer);
      preempted_server->running = FALSE;
      preempted_server->remain_budget -= (jiffies-preempted_server->
timestamp);


      if(preempted_server->active_task != NULL){   // means task is
running in that server
           preempted_task = preempted_server->active_task;
           // try to run preemted task on other servers of its
cluster
           if(preempted_cluster->nr_active_servers !=0){
               // there are other servers from this cluster running
try to run there
               if((try_to_run_task(preempted_task,
preempted_cluster))== FALSE){
                         FLAG = TRUE;
               }else{
                         //vchsf_job_start(preempted_task->rt);
                   }
           }else{
                  FLAG = TRUE;
           }
           if(FLAG == TRUE){   // means you need to dequeue that task
and put it into cluster ready_queue
                    #ifdef WITH_TASKS
                //dequeue_task(preempted_task->rt);
                preempted_task->rt->task->state =
TASK_UNINTERRUPTIBLE;
                  set_tsk_need_resched(preempted_task->rt->task);
                #endif
                     bitmap_insert_tpq(&preempted_cluster->
READY_TASKS, preempted_task);
                     printk(KERN_WARNING "job queued in preempt
server: %d %s %d %d %d (%lu)\n", preempted_task->rt->pid,
preempted_task->rt->task->comm, preempted_task->rt->
prio,preempted_task->rt->cpu_id, preempted_task->rt->server_id,
jiffies);

           }
      }

        if(preempted_server->remain_budget == 0){
           stop_server(preempted_server);
           printk(KERN_WARNING "server_complete: server:%d cluster:%d
cpu:%d (%lu)\n",preempted_server->id, preempted_server->mycluster->
vc_cluster_id, preempted_server->server_cpu, jiffies);
        }else{
           printk(KERN_WARNING "server_preempted: server:%d cluster:%
d remaining budget:%d cpu:%d (%lu)\n",preempted_server->id,
preempted_server->mycluster->vc_cluster_id, preempted_server->
remain_budget, preempted_server->server_cpu,  jiffies);
               preempted_cluster->nr_ready_servers++;  // earlier it
was common
               bitmap_insert_spq(&SERVER_READY_QUEUE,
preempted_server);    // suspect
        }

      server->mycluster->nr_ready_servers--;
      server->mycluster->nr_active_servers++;
      if(server->mycluster->cluster_state != CLUSTER_ACTIVE){
            server->mycluster->cluster_state = CLUSTER_ACTIVE;
      }

      server->timestamp = jiffies;
      server->budget_expiration_time = jiffies+ server->
```

```
remain_budget;
      server->running = TRUE;
        server->server_cpu = cpu;    // latest detected bug


      if((running_task = bitmap_retrieve_tpq(&server->mycluster->
READY_TASKS))==NULL){
        //if((running_task = bitmap_retrieve_tpq(&READY_TASKS
[server->mycluster->vc_cluster_id]))==NULL){
            // means there is no more ready task jobs in cluster task
queue so remain idle
            server->active_task = NULL;
                //idle_cpus++;
      }else{
            server->active_task = running_task;

            rt = running_task->rt;
                rt->server_id = server->id;
                #ifdef WITH_TASKS
                migrate_vchsf_task(rt, cpu);
                if(rt->task->state != TASK_RUNNING){
                wake_up_process(rt->task);
                }
            //migrate_task(resch_task_t *rt, int cpu_dst)
                //rt->server_id = server->id;
                //enqueue_task(rt);
            //migrate_task(rt, cpu);
                //vchsf_job_start(rt);
                #endif
                printk(KERN_WARNING "job running in preempt server: %
d %s %d cpu:%d server:%d cluster:%d (%lu)\n", rt->pid, rt->task->
comm, rt->prio, rt->cpu_id, rt->server_id, rt->cluster_id, jiffies);
      }
      // change cpu status
      cpu_status[cpu].active_server = server;
      cpu_status[cpu].active_cluster_priority = server->mycluster->
priority;
        cpu_status[cpu].cpu_busy = CPU_BUSY;   // to indicate server
is running there

    //insert_timer(&server->server_budget_timer,
server_complete_handler, (unsigned long)server, server->
budget_expiration_time);

        printk(KERN_WARNING "server preemption: preempter server:%d
preempter cluster:%d cpu:%d (%lu)\n",server->id, server->mycluster->
vc_cluster_id, server->server_cpu, jiffies);
      printk(KERN_WARNING "server preemption: preempted server:%d
preempted cluster:%d cpu:%d (%lu)\n",preempted_server->id,
preempted_server->mycluster->vc_cluster_id, server->server_cpu,
jiffies);
        setup_timer_on_stack(&server->server_budget_timer,
server_complete_handler, (unsigned long)server);
      mod_timer(&server->server_budget_timer, server->
budget_expiration_time);

      return TRUE;
}
//stub
/*
 * 1. try to run the server in a idle processor
 * 2. no idle processor, then find a processor running server from
low priority cluster
 * 3. if 2 is true then prempt that server
 * 4. server and tasks inserting
 * 5. update server and cluster status
 *
 */
int try_to_run_server(server_t *server){

      int cpu,i, lowest_priority, preempted_cpu;

      if(idle_cpus !=0){
         cpu = check_idle_cpus();

         if((run_server(server, cpu))== TRUE){
             return TRUE;
         }else{
             return FALSE;
         }
      }else{   // try to find a server to preempt
       for(i=0; i< NR_RT_CPUS; i++){
         if(i == 0){
             lowest_priority = cpu_status[i].active_cluster_priority;
             preempted_cpu = i;
         }else if(lowest_priority < cpu_status
[i].active_cluster_priority){
               lowest_priority = cpu_status
[i].active_cluster_priority;
               preempted_cpu = i;
         }
       }
       if(lowest_priority > server->priority){
               //printk(KERN_WARNING "server preemption should be:
server:%d cpu:%d (%lu)\n",server->id, cpu, jiffies);
               //return FALSE;   // extra line for debugging

             if((preempt_server(server, preempted_cpu))== TRUE){
               return TRUE;
             }else{
               return FALSE;
             }

       }else{
            return FALSE;
       }
      }
      // requires to program budget expiration timer
      //return TRUE;
}

void update_lowest_cluster_prio(void){

      int i;

      for(i = 0; i < NR_OF_CLUSTERS; i++){
         if(CLUSTERS[i].cluster_state == CLUSTER_ACTIVE){
             if(CLUSTERS[i].priority > lowest_active_cluster_prio){
                   lowest_active_cluster_prio = CLUSTERS[i].priority;
             }
         }
      }
}


/*
 *
 * Function for assigning server budget statically to the servers of
cluster
 *
 */
void assign_server_budget(cluster_t *cluster, int nr_cpus){
      int i, total_budget, assign_budget;

      if(cluster->remain_budget == 0){
            return;  // no budget to assign
      }else{
            total_budget = cluster->remain_budget;
               if(nr_cpus > 1){
            assign_budget = (total_budget/nr_cpus) + 1;
               }else{
                assign_budget = total_budget;
               }
            for(i = 0; i < nr_cpus; i++){
          if((total_budget - assign_budget) <= 0){
             cluster->SERVERS[i].budget = total_budget;
                 cluster->SERVERS[i].remain_budget = total_budget;
                 printk(KERN_WARNING "server received budget:
server:%d budget:%d remain_budget:%d (%lu)\n",cluster->SERVERS[i].id,
cluster->SERVERS[i].budget, cluster->SERVERS[i].remain_budget,
jiffies);
          }else{
                   cluster->SERVERS[i].budget = assign_budget;
                    cluster->SERVERS[i].remain_budget =
assign_budget;
                   total_budget -= assign_budget;
          }
            }
      }
}


/*
 *
 * Cluster release handler
 *
 */


void cluster_release_handler(unsigned long __data) {

      // needs global spinlock

      static int FIRST_RUN = TRUE;    // static local flag variable
to indicate first run only initialised once
      static unsigned long start = 0;  // starting time of the system
      cluster_t *released_cluster = (cluster_t *)__data;
      cluster_t *temp_cluster = NULL;
      cluster_t *peek = NULL;
      int i,j, next_release_event, event;
      int multi_release = 0;   // multiple release flag
        struct timespec tmp_time_start, tmp_time_end, overhead;
      relNode *node;
      relNode *Released[MAX_NR_OF_CLUSTERS];     // array for
handling multiple release
      unsigned long flags;

      spin_lock_irqsave(&global_lock, flags);

      getnstimeofday(&tmp_time_start);
        //preempt_disable();

      // there is an error in timing so quit
      if ((jiffies - start) > RUN_JIFFIES && start != 0) {
            //destroy_timer_on_stack(&event_timer);
            remove_timer(&event_timer);
              //preempt_enable();
            //destroy_timer_on_stack(&event_timer);
            spin_unlock_irqrestore(&global_lock, flags);
            return;
      }

      if (released_cluster == NULL) { // EXTRA
            printk(KERN_WARNING "(cluster_release_handler)
cluster==NULL???\n"); // EXTRA
                //preempt_enable();
            spin_unlock_irqrestore(&global_lock, flags);
            return; // EXTRA
      }

      released_cluster->remain_budget = released_cluster->budget;
      released_cluster->cluster_state = CLUSTER_READY;

        printk(KERN_WARNING "cluster_release: cluster:%d
\n",released_cluster->vc_cluster_id);

      // Insert cluster in ready queue...
      bitmap_insert(&CLUSTER_READY_QUEUE, released_cluster);
      // Deallocate the timer...
      //destroy_timer_on_stack(&timer);
      remove_timer(&event_timer);
      //destroy_timer_on_stack(&event_timer);

    if(FIRST_RUN == TRUE){    // this is the first run by any cluster
and so 1. all processor idle 2. this is the highest prio cluster
        FIRST_RUN = FALSE;
        start = jiffies;
        idle_cpus = NR_RT_CPUS;
        released_cluster = bitmap_retrieve(&CLUSTER_READY_QUEUE);
        lowest_active_cluster_prio = released_cluster->priority;
        released_cluster->remain_budget = released_cluster->budget;
        assign_server_budget(released_cluster, released_cluster->
nr_cpus);    // stub
        released_cluster->cluster_state = CLUSTER_ACTIVE;
        printk(KERN_WARNING "inside cluster released: first block
\n");
        printk(KERN_WARNING "cluster starts running: cluster:%d (%lu)
\n",released_cluster->vc_cluster_id, jiffies);
        printk(KERN_WARNING "cluster release idle cpus before:%d
\n",idle_cpus);
        for(i = 0; i < released_cluster->nr_cpus; i++){
              if((try_to_run_server(&released_cluster->SERVERS[i]))==
TRUE){
                          released_cluster->SERVERS[i].running =
TRUE;
                    released_cluster->nr_active_servers++;
              }else{
                          released_cluster->SERVERS[i].running =
FALSE;
                         bitmap_insert_spq(&SERVER_READY_QUEUE,
&released_cluster->SERVERS[i]);
                    released_cluster->nr_ready_servers++;
              }
        }

        printk(KERN_WARNING "cluster_release idle cpus after:%d
\n",idle_cpus);
        if(idle_cpus != 0){     // means other clusters may run

            while(idle_cpus !=0){
                    if((temp_cluster = bitmap_retrieve
(&CLUSTER_READY_QUEUE)) == NULL){
                          printk(KERN_WARNING "\n No more clusters,
processors idle");
                         break;
                    }else{
                         temp_cluster->cluster_state =
CLUSTER_ACTIVE;
                         temp_cluster->remain_budget = temp_cluster->
budget;
                         printk(KERN_WARNING "cluster starts running
first block extra: cluster:%d (%lu)\n",temp_cluster->vc_cluster_id,
jiffies);
                         if(temp_cluster->priority >
lowest_active_cluster_prio){
                              lowest_active_cluster_prio =
temp_cluster->priority;
                         }
                         assign_server_budget(temp_cluster,
temp_cluster->nr_cpus);
                         for(j =0; j < temp_cluster->nr_cpus; j++){
                               if((try_to_run_server(&temp_cluster->
SERVERS[j]))== TRUE){
                                      temp_cluster->SERVERS
[j].running = TRUE;
                                      temp_cluster->
nr_active_servers++;
                               }else{
                                     temp_cluster->SERVERS[j].running
= FALSE;
                                      bitmap_insert_spq
(&SERVER_READY_QUEUE, &temp_cluster->SERVERS[j]);

                                      temp_cluster->
nr_ready_servers++;
                               }

                         }

                    }
               }
          }

          // there can be other clusters which are ready to run not
necessary
          /*
          while((temp_cluster = bitmap_retrieve(&CLUSTER_READY_QUEUE))
!= NULL){
               // make servers of the clusters ready to run
               for(j =0; j < temp_cluster->nr_cpus; j++){
                      temp_cluster->READY_SERVERS[j]= temp_cluster->
SERVERS[j];
                      temp_cluster->nr_ready_servers++;
               }
               if(temp_cluster->cluster_state != CLUSTER_READY){
                     temp_cluster->cluster_state = CLUSTER_READY;
               }
          }
          */
          // timer for next scheduling event of cluster release
          next_release_event = 0;
          relPq_retrieve(&CLUSTER_RELEASE_QUEUE, &next_release_event);
          node = relPq_retrieve(&CLUSTER_RELEASE_QUEUE,
&next_release_event); // We need the node later but...
          relPq_insert(&CLUSTER_RELEASE_QUEUE, next_release_event,
node);

    }else{  // this not the first time run
```

```
      // update the release queue for next run
        printk(KERN_WARNING "inside cluster released: later block
\n");
      event = 0;
      multi_release = 0;
      // retrieve the currently released cluster from release queue
        relPq_retrieve(&CLUSTER_RELEASE_QUEUE, &event);
        node = relPq_retrieve(&CLUSTER_RELEASE_QUEUE, &event);

            relPq_peek(&CLUSTER_RELEASE_QUEUE, &i); // Check the
value of the second element in the release queue

            if (i != event) { // Only one cluster to release in this
point.....
                  released_cluster = bitmap_retrieve
(&CLUSTER_READY_QUEUE);
                  // insert the currently released cluster for its
next release
                  relPq_insert(&CLUSTER_RELEASE_QUEUE, event+CLUSTERS
[node->index].period, node);
                  // Fetch next release event from other release
events in the release queue
                  event = 0;
                  relPq_retrieve(&CLUSTER_RELEASE_QUEUE, &event);
                  node = relPq_retrieve(&CLUSTER_RELEASE_QUEUE,
&event);
                  relPq_insert(&CLUSTER_RELEASE_QUEUE, event, node);
//?
                  next_release_event = event -
CLUSTER_RELEASE_QUEUE.virtual_time;
            }
            else { // More than one cluster has to be released...
                  //multi_release = 1;
                  next_release_event = event;
                  Released[0] = node;     // means the node released
with this call
                  i = 1;
                  while (TRUE) {
                        event = 0;
                        // retrieve other release events in same time
from release queue
                        relPq_retrieve(&CLUSTER_RELEASE_QUEUE,
&event);
                        Released[i] = relPq_retrieve
(&CLUSTER_RELEASE_QUEUE, &event);
                        // Refill budget
                        CLUSTERS[Released[i]->index].cluster_state =
CLUSTER_READY;
                        CLUSTERS[Released[i]->index].remain_budget =
CLUSTERS[Released[i]->index].budget;
                        // Insert the cluster in the ready queue
                        bitmap_insert(&CLUSTER_READY_QUEUE, &
(CLUSTERS[Released[i]->index]));
                        i++;
                        relPq_peek(&CLUSTER_RELEASE_QUEUE, &event);
                        if (event != next_release_event || event < 0)
                              break;
                  }
                  temp_cluster = bitmap_get(&CLUSTER_READY_QUEUE); //
Now lets see who has highest prio...
           /* if(temp_cluster->priority < released_cluster->priority)
{  // one of the released cluster has higher priority
                  temp_cluster = bitmap_retrieve
(&CLUSTER_READY_QUEUE);
                  bitmap_insert(&CLUSTER_READY_QUEUE,
&released_cluster);
```

```
                  released_cluster = temp_cluster;
            }*/
                  for (j = 0; j < i; j++) {
                        if (temp_cluster->vc_cluster_id == Released
[j]->index) { // Ok, one of the newbies is highest!
                              released_cluster = bitmap_retrieve
(&CLUSTER_READY_QUEUE);
                              break;
                              //special = 2;
                        }
                  }
                  // Now update and insert all elements in the
release queue...
                  for (j = 0; j < i; j++) {
                        relPq_insert(&CLUSTER_RELEASE_QUEUE,
next_release_event+CLUSTERS[Released[j]->index].period, Released[j]);
                  }

                  multi_release = i -1;   // number of released used
later excluding the main released one
                  // Fetch next release event...
                  event = 0;
                  relPq_retrieve(&CLUSTER_RELEASE_QUEUE, &event);
                  node = relPq_retrieve(&CLUSTER_RELEASE_QUEUE,
&event);
                  relPq_insert(&CLUSTER_RELEASE_QUEUE, event, node);
                  next_release_event = event -
CLUSTER_RELEASE_QUEUE.virtual_time;
            }

         if((idle_cpus > 0) || (released_cluster->priority <
lowest_active_cluster_prio)){
         released_cluster->remain_budget = released_cluster->budget;
       assign_server_budget(released_cluster, released_cluster->
nr_cpus);
       released_cluster->cluster_state = CLUSTER_ACTIVE;
         printk(KERN_WARNING "cluster starts running: cluster:%d (%
lu)\n",released_cluster->vc_cluster_id, jiffies);
        // schedule clusters on the cpus
         for(j = 0; j < released_cluster->nr_cpus; j++){
                    if((try_to_run_server(&released_cluster->SERVERS
[j]))== TRUE){
                            released_cluster->SERVERS[j].running =
TRUE;
                            released_cluster->nr_active_servers++;
                    }
                      if(released_cluster->SERVERS[j].running != TRUE)
{
                         released_cluster->SERVERS[j].running =
FALSE;
                         bitmap_insert_spq(&SERVER_READY_QUEUE,
&released_cluster->SERVERS[j]);
                         released_cluster->nr_ready_servers++;
                      }
            }
                if(released_cluster->priority >
lowest_active_cluster_prio){
                     lowest_active_cluster_prio = temp_cluster->
priority;
                }else{
                  update_lowest_cluster_prio(); // this means there
is a prememption of cluster
                }
         }else{              // cluster remains ready in ready queue
               bitmap_insert(&CLUSTER_READY_QUEUE, released_cluster);
         }
       if(multi_release > 0){  // there is more than one cluster
released which can run
              while(multi_release !=0){           // wrong here needs
to be corrected should be same as earlier idle_cpus
                   peek = bitmap_get(&CLUSTER_READY_QUEUE);
                   if((idle_cpus !=0) ||(peek->priority <
lowest_active_cluster_prio)){
                     if((temp_cluster = bitmap_retrieve
(&CLUSTER_READY_QUEUE)) == NULL){
                          printk(KERN_WARNING "\n No more clusters,
processors idle (%lu)", jiffies);
                          break;
                     }else{
                         temp_cluster->cluster_state =
CLUSTER_ACTIVE;
                         temp_cluster->remain_budget = temp_cluster->
budget;
                         assign_server_budget(temp_cluster,
temp_cluster->nr_cpus);
                         printk(KERN_WARNING "cluster starts running:
cluster:%d (%lu)\n",temp_cluster->vc_cluster_id, jiffies);
                         for(j =0; j < temp_cluster->nr_cpus; j++){
                                 if((try_to_run_server
(&temp_cluster->SERVERS[j]))== TRUE){
                                    temp_cluster->SERVERS[j].running
= TRUE;

                                    temp_cluster->
nr_active_servers++;
                                 }
                               if(temp_cluster->SERVERS[j].running !=
TRUE){
                                temp_cluster->SERVERS[j].running =
FALSE;
                                bitmap_insert_spq
(&SERVER_READY_QUEUE, &temp_cluster->SERVERS[j]);
                                temp_cluster->nr_ready_servers++;
                               }
                       }
                         update_lowest_cluster_prio();
                         multi_release--;  // release of the cluster
complete
                     }  // end of else

                   }else{
                         break;
                   }

              } // end of while
          }
    }

    // next release evet should be scheduled here
   // insert_timer(&event_timer, cluster_release_handler, (unsigned
long)&CLUSTERS[node->index], (jiffies+next_release_event));
    setup_timer_on_stack(&event_timer, cluster_release_handler,
(unsigned long)&CLUSTERS[node->index]);
    mod_timer(&event_timer, (jiffies+next_release_event));


    CLUSTER_RELEASE_QUEUE.virtual_time =
CLUSTER_RELEASE_QUEUE.virtual_time + next_release_event;
    //preempt_enable();
    getnstimeofday(&tmp_time_end);
    overhead = timespec_sub(tmp_time_end, tmp_time_start);
    total_overhead += overhead.tv_nsec + overhead.tv_sec*1000000000;
    spin_unlock_irqrestore(&global_lock, flags);
```

```
}

/*
 * THis is the handler called when a cluster completes its allocated
budget in a peiord
 * this one is not related to timer, called by the server complete
handler when the number of
 * active server in the cluster becomes zero
 *
 */

void cluster_complete_handler(cluster_t* complete){

      server_t *to_run_server;
      cluster_t *temp_cluster;
      int j;

      // cluster_t should be cleared
      complete->cluster_state = CLUSTER_EXPIRED;  // running flag to
indicate cluster is running
      if(complete->nr_active_servers != 0){
            printk(KERN_WARNING "\n Servers are not released
properly");
            //complete->nr_active_servers = 0;
      }
      if(complete->nr_ready_servers != 0){
            printk(KERN_WARNING "\n Servers are not handled
properly");
      }
      if(complete->remain_budget != 0){
            printk(KERN_WARNING "\n Server budget not consumed
properly");
      }
      /*
      if(bitmap_get(&complete->ACTIVE_SERVERS)!= NULL){
            printk(KERN_WARNING "\n Still active servers in the
cluster");
            return;
      }

      if(bitmap_get(&complete->READY_SERVERS)!= NULL){
            printk(KERN_WARNING "\n Still ready servers in the
cluster");
            return;
      }
      */
      complete->nr_active_servers = 0;
      complete->nr_ready_servers = 0;
      complete->remain_budget = 0;

        printk(KERN_WARNING "cluster_complete: cluster:%d (%lu)
\n",complete->vc_cluster_id, jiffies);

    if(cluster_complete_on_preemption == TRUE){
       cluster_complete_on_preemption = FALSE;
       update_lowest_cluster_prio();
       return;
    }

      // here idle cpu count should !=0
      // check for a active cluster which has ready servers
    while(idle_cpus != 0){
      if((to_run_server = check_active_clusters_for_servers()) ==
NULL){
            break;
```

```
          }else{
                    if((try_to_run_server(to_run_server))== TRUE){
                            temp_cluster = to_run_server->mycluster;
                            to_run_server->running = TRUE;
                            temp_cluster->nr_active_servers++;
                    }
          }
    }

      // check for a ready cluster and launch its servers
    while(idle_cpus != 0){
      if((temp_cluster = bitmap_retrieve(&CLUSTER_READY_QUEUE)) ==
NULL){
            // there is no more ready clusters in the system to
launch
            printk(KERN_WARNING "NO Cluster in ready queue\n");
            break;
      }else{
             temp_cluster->cluster_state = CLUSTER_ACTIVE;
             temp_cluster->remain_budget = temp_cluster->budget;
             assign_server_budget(temp_cluster, temp_cluster->
nr_cpus);
             printk(KERN_WARNING "cluster starts running: cluster:%d
(%lu)\n",temp_cluster->vc_cluster_id, jiffies);
             for(j =0; j < temp_cluster->nr_cpus; j++){
                     if((try_to_run_server(&temp_cluster->SERVERS
[j]))== TRUE){
                        temp_cluster->SERVERS[j].running = TRUE;

                        temp_cluster->nr_active_servers++;
                     }

                 if(temp_cluster->SERVERS[j].running != TRUE){
                      temp_cluster->SERVERS[j].running = FALSE;
                      bitmap_insert_spq(&SERVER_READY_QUEUE,
&temp_cluster->SERVERS[j]);
                      temp_cluster->nr_ready_servers++;

                 }
            }
             update_lowest_cluster_prio();  // updating lowest
cluster priority
      }
    }
    update_lowest_cluster_prio();

}


static int __init vchsf_init(void)
{

      cluster_t *highest_prio_cluster;  // number of servers that
ready to run initially
      int i;


      spin_lock_init(&global_lock);
      total_overhead = 0;
      init_timer(&event_timer);
      init_pq(&CLUSTER_READY_QUEUE);
      init_spq(&SERVER_READY_QUEUE);
      cluster_complete_on_preemption = FALSE;

      printk(KERN_INFO "VC-HSF: HELLO!\n");
    // initialiasing global list of cpus status as idle
      for(i =0; i < NR_RT_CPUS; i++){
            cpu_status[i].cpu_busy = CPU_IDLE;
            cpu_status[i].active_cluster_priority = 0;
            cpu_status[i].active_server = NULL;
        }

      // initialize cluster with cluster_t ,c_id, nr_servers,
budget_msecs, period_msecs, priority
      //init_cluster(&CLUSTERS[0],0, 2, 10, 16, 0);
      //init_cluster(&CLUSTERS[1],1, 1, 20, 32 , 1);
        init_cluster(&CLUSTERS[0],0, 2, 10, 12, 0);
        init_cluster(&CLUSTERS[1],1, 2, 20, 32, 1);


      /**************** Put the all the clusters in the cluster ready
queue ****************/
      for (i = 0; i < NR_OF_CLUSTERS; i++) {
            bitmap_insert(&CLUSTER_READY_QUEUE, &CLUSTERS[i]);
      }


      if ( (highest_prio_cluster = bitmap_retrieve
(&CLUSTER_READY_QUEUE)) == NULL) {
            printk(KERN_WARNING "VC-HSF (hsf_init): Server ready
queue empty!!!\n");
            return 0;
      }


      // Initialize nodes that will reside in the cluster release
queue structure...
      for (i = 0; i < NR_OF_CLUSTERS; i++) {
            RelNodes[i].index = i;
            RelNodes[i].next = NULL;
      }

      // Initialize cluster release queue
      if ( (relPq_init(&CLUSTER_RELEASE_QUEUE, find_largest_period
(CLUSTERS, NR_OF_CLUSTERS) )) < 0) {
            printk(KERN_WARNING "'relPq_init' failed!!!\n");
            return 0;
      }

      // Insert the cluster release queue nodes...
      for (i = 0; i < NR_OF_CLUSTERS; i++) {
            relPq_insert(&CLUSTER_RELEASE_QUEUE, CLUSTERS[i].period,
&RelNodes[i]);
      }

      setup_timer_on_stack(&event_timer, cluster_release_handler,
(unsigned long)highest_prio_cluster);
      mod_timer(&event_timer, (jiffies+msecs_to_jiffies
(SYSTEM_TIMEOUT))); // Start in 10 seconds...
      //insert_timer(&event_timer, cluster_release_handler, (unsigned
long)highest_prio_cluster, (jiffies+msecs_to_jiffies
(SYSTEM_TIMEOUT)));

      printk(KERN_WARNING "VC-HSF: HELLO! (%lu)\n",
(jiffies+msecs_to_jiffies(SYSTEM_TIMEOUT)));

      // Install our plugins...
      install_scheduler(task_run, NULL, job_release, job_complete);
// these corresponds to task run, job release and job complete plugin
of core
```

```
      return 0;
}







static void __exit vchsf_exit(void)
{
        printk("total overhead(nano_sec): %lu\n", total_overhead);
      printk(KERN_INFO "VC-HSF: GOODBYE!\n");
      uninstall_scheduler();
}


module_init(vchsf_init);
module_exit(vchsf_exit);
```