Designing and Integrating New Scheduler For MOOL

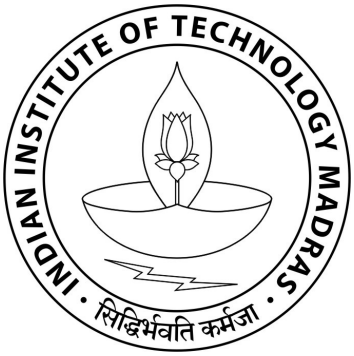
A Project Report

submitted by

AMRITENDU MONDAL

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.

May 2014

THESIS CERTIFICATE

This is to certify that the thesis entitled **Designing and Integrating New Scheduler For MOOL**, submitted by **Amritendu Mondal**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bona fide record of the research work carried out by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. D. Janakiram
Research Guide
Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date:

ACKNOWLEDGEMENTS

Master of technology has been a very en-lighting experience for me. I would like to take this opportunity to acknowledge all the individuals who helped me at various stages in this journey.

I am extremely thankful to my guide Prof. D.Janakiram who helped me by giving constant guidance throughout my tenure at IIT Madras. He has spent a lot of his valuable time with me in debugging and solving issues in my project. I would always cherish all the time I used to discuss with him about the new object oriented Linux MOOl and the fundamentals of Linux kernel. I shell always appreciate his suggestion and views on our social improvement. He made me realize how a teacher should work with a student in a friendly manner so that both the students and the teacher can be happy through out the journey. I would like to acknowledge the CDAC kernel development team who helped me in every steps of my project.

I would like to thank my lab mate S J Balaji, who was a backbone of my project and helped me to work with the GPU assisted scheduler implemented by him. I would also like to thank others lab members,specially Sriram who helped me in every step. I would also like to thank DOS Lab for giving me a nice environment to work in. I would always agree and remember the facility I got here. I would remember IIT Madras for all their support and facility.

Finally I would like to express my most sincere to my parents. My journey couldn't have accomplished without their love and best wishes.

ABSTRACT

KEYWORDS: GPU, Linux Kernel, Scheduler.

MOOL is a Linux kernel distribution where one can write device drivers in C++. This provides better extensibility and maintainability as compared to developing them in C. In phase-I, we re-engineered an object oriented device driver for bluetooth. The code-base of bluetooth is fast evolving as new functionalities are added. We show that this object oriented bluetooth device driver has many advantages over the C style device driver code.

As multi-core and many-core systems are becoming common-place, how to efficiently use them is being widely researched. Newer scheduling policies are being developed. In order to integrate them easily into Linux, we propose an object oriented design for Linux scheduler. We introduced object-oriented concepts in Complete-Fair-Scheduler (CFS) code, an existing Linux scheduler implemented in C. This re-factored CFS, now implemented in C++, is tested in MOOL. We noticed a minimal performance overhead compared to the original CFS.

In many core systems the scheduling depends on various parameters like topology, inter core component utilization, etc.. to resolve heating issues due to non-uniform power consumption across cores. GAS (GPU assisted scheduler), developed by S J Balaji in DOS lab, uses GPU to analyse the various scheduling parameters and decides an optimal placement of processes across cores. GAS measures a stability metric to decide whether optimization algorithm needs to be

run. GAS is implemented entirely in user-space, it uses the perf utility to gather parameters from the kernel and check stability. This repeatedly invokes the perf utility and sets the shared memory for GAS. We moved the stability check into the kernel space by adding KGAS kernel module that checks the stability threshold. We present results from running our GAS implementation on Hack-bench workloads.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

IITM	Indian Institute of Technology, Madras
GPU	Graphics processing unit
GAS	GPU assisted scheduler
CFS	Completely fair scheduler
SD	Scheduling domain
CPU	Central processing unit

CHAPTER 1

Introduction

MOOL is a Linux kernel distribution which allows writing device driver in C++. MOOL is a redesigned of existing Linux kernel with minimal object-oriented components. MOOL provides a object-oriented wrapper,which supports the development of Linux kernel subsystem in Object-oriented fashion. MOOL performance is almost same as existing C kernel.

Bluetooth is a widely use technology. It is meant for exchanging informations over short distance. Not only mobile, but also in the personal computers, in various embedded systems we find many bluetooth applications. Bluetooth works in a master slave fashion. Bluetooth is a layered protocol architecture. Important protocols are LMP,L2CAP,AMP etc..

Object oriented device drivers are preferable over C implemented device driver. It allows object oriented design at language level. An object oriented driver gives good maintainability over a C style code. In Phase-I we re-engineered the bluetooth device driver code. Bluetooth devices are manufactured by different vendors.The controller stack is vendor specific ,hence different vendors need different code base ,which is repetitive work. We propose a better design of bluetooth controller stack,which allow to reuse common functionalities. This is a group project with Deepankar Patra(CS12M015) and Mahesh Gupta(CS12M018).

Multi-core is a computing component where there is more than one central processing unit. Those independent CPUs can execute different instructions at the

same time. Multi-core is becoming a common-place. C implemented Linux kernel was designed for a single core system. Later Symmetric Multi-Processing(SMP) is added to it. Current kernel code use multi-core but it does not use all the core in a optimal way.

To use all the CPUs in an optimal way we need a new scheduler design. We propose an object-oriented Linux scheduler. Our current new scheduler policy is CFS. This new scheduler has a base class ,which captures all the common functionalities. New classes reuses those functionalities and overrides its own algorithms . We implemented this new design and tested it in MOOL. Debugging and solving various issues for our multi-core scheduler was a bit challenging. We used printk for debugging the kernel. We also used gdb for some cases. Sometimes those debugging technique are not helpful,like inside the IRQ routine or inside the schedule function. We evaluated our new scheduler in sysbench workload and We noticed a minimal performance overhead compared to the original CFS.

Many-core systems are becoming affordable for personal desktop. The study shows that current process scheduling techniques are not optimal for many-core system. In many-core system spatial scheduling is a good choice over temporal scheduling technique.In many-core systems the scheduling depends on various parameters like process topology,inter component utilization, etc. We propose a GPU assisted scheduler to analyse the various scheduling parameters and decides an optimal placement of processes across cores.

GPU assisted scheduler is developed by S J Balaji, DOS Lab. We integrated it in MOOL kernel. GAS measures a stability metric to decide whether optimization algorithm needs to be run. We calculate that stability value in a kernel module. The kernel module calculates the stability value and writes it into the proc file

system. GAS is entirely in user-space ,hence it reads the stability value and call GAS.We evaluated this integrated scheduler in hack-bench workload.

1.1 Contributions

The project contribution are as follows.

- In PHASE-I we designed and re-engineered an object oriented bluetooth device driver in C++.
- In PHASE-II we did some literature survey on Linux scheduler and integrated GAS in MOOL.
- In PHASE-III we designed an object-oriented scheduler and evaluated GAS on that.

1.2 Organization of The Thesis

In this section, we discuss about the content of the thesis. We discuss our phase-I work, an object-oriented bluetooth device driver in part-I. Then we discuss some background work on Linux scheduler in chapter 3. In chapter 4, we discuss about multi-core object oriented Linux scheduler. Then in part-III, we introduce GPU assisted scheduler scheduler(GAS), developed by S J Balaji. In this chapter we also discuss about integrating GAS in the MOOL.

Part I

AN OBJECT ORIENTED DEVICE DRIVER FOR BLUETOOTH

CHAPTER 2

Re-engineering Object-oriented Bluetooth Device Driver

2.1 Introduction

Bluetooth hardware are being manufactured by various vendors. Controller stack is hardware specific,hence different device driver code is required for different vendors. Writing the device driver code for different vendors becomes a repetitive work. Writing code for each vendors and maintaining those create overheads.

The code-base of bluetooth is fast evolving as new functionalities are added. An object-oriented(OO) device driver provides better extensibility and maintainability as compared to developing them in C. In OO fashioned design we write a base class and implements the common functionalities. Each manufacturer can inherit the common methods and override the device specific methods. Benefit of writing an object-oriented bluetooth driver is follows.

- Easily maintainable code.
- Easily understandable code.
- Adding new features are not much overhead.

2.2 Bluetooth Architecture

We worked with bluetooth version 3.10. Bluetooth architecture is divided into two parts. One is controller stack and other one is host stack. Controller stack is embedded in the device. It is device specific. The host stack is OS specific. Bluetooth applications communicate with the host stack.

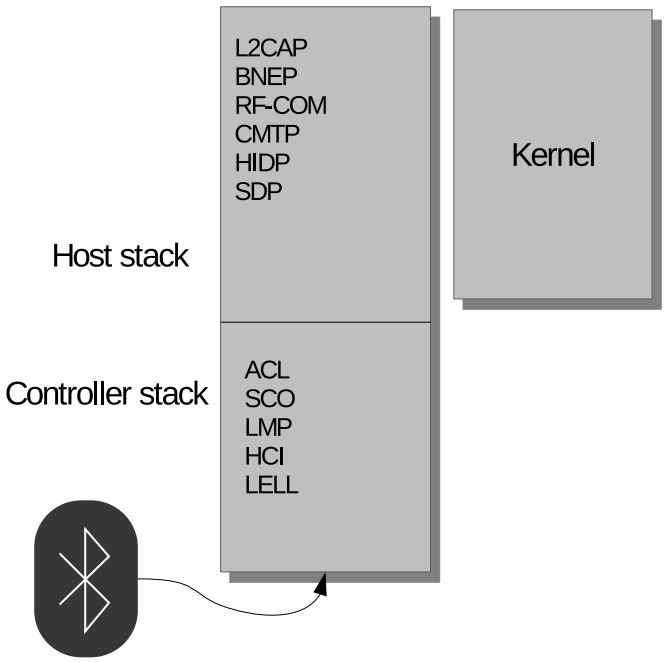


Figure 2.1: Bluetooth architecture

The above figure2.1 represents the bluetooth architecture . The control stack contains various components like ACL (Asynchronous connectionless link),SCO (Synchronous Connection Oriented link),LMP (Link Management Protocol),HCI

(Host Controller Interface),LELL (Low Energy Link Layer). Whereas the host stack contains various protocol like L2CAP (Logical Link Controller and Adaptation Protocol),BNEP (Bluetooth Network Encapsulation Protocol),RFCOMM (Radio Frequency Communication),CMTP (CAPI Message Transport Protocol),HIDP (Human Interface Device Profile),SDP (Service Discovery Protocol). Here L2CAP delivers the packets to the HCI device or in a host less system to the lick manager ACL. BNEP is bounded to L2CAP and used for delivering networks packets on the top of L2CAP. RFCOMM is basically a radio frequency communication , which is again bound in L2CAP and provides data streams to the user.

2.3 Generics Steps

We followed the below mentioned steps to convert the whole code into C++.

- Find out a stable version of bluetooth and checks the availability of code with a proper documentation.
- Read the code base and understand it properly for a good design.
- Compile the code and generate the kernel object files.
- Insert the module using insmode or modprobe. This might create problem sometimes if the dependent drivers like USB are not installed.
- One can enable those module in the kernel configuration file and compile and install the kernel with the downloaded module. This ensures that our bluetooth code is working as a kernel module, and we can work with this code base.
- Then start implementing the OO code of the tested C implemented code.
- One can start by creating the base class and the inheriting the methods and putting the implementation in the subclass.
  - While working with a C++ code base, one might face the issues of name-mangling.
  - The call back functions which are registered in the kernel can not be put inside a class.
  - Those can be put inside a extern ”C” block.

- Finally compile and install the C++ code base.
  - We can compile it separately , but we need to put the required library and header files correctly.
  - We can attach it with the linux kernel code and compile kernel and install it in a system.

2.4 Requirements

To start writing a device driver in MOOL, one should meet the following system requirements.

- MOOl kernel source code, this can be downloaded online.
- Modified gcc compiler. MOOL researcher team has changed the default gcc to support,C++ code in Linux kernel. One should take the code of this changed GCC and install and test it before compiling a C++ driver.
- We used gcc version 4.4.
- Modified g++, for the same reason one has to download and install the modified g++. We have worked with version 4.4.

2.5 Implementation

We worked with MOOL version 3.6. MOOl-3.6 supports bluetooth version 2.1. We made the required changes to support the bluetooth version 3.10. In section 3.2, we discuss some of the issues we faced while doing the integration. The controller stack of bluetooth is device specific and very few abstraction is possible, whereas in the host interface abstraction is possible.

L2CAP protocol is written two C files.l2cap_sock.c deals with the socket managements and l2cap_core.c deals with the core functionalities. Core functionalities mean channel management, connection management,packet sequencing, service

level security,signalling, core HCI related functionalities. All of these functionalities are implemented as a set of functions and there is a clear separation regarding how these functionalities interact. We have created a base class as l2cap_core. Inheriting the base class we creates subclasses to separate the functionalities and finally implements specific class methods in child classes.

AMP (Alternative MAC/PHY) is a improvement over 802.11 for providing high speed transport between bluetooth devices. We show a class diagram of AMP protocol in figure 2.2.AMP protocol is implemented in amp.c file. AMP has two interfaces one is controller and another one is physical link interface. The controller interface controls the various operations. The physical link interface is responsible for adding , removing physical links.We make these two interfaces separate for better maintainability. The functionalities are divided into two classes. The controller interface is named as AMP_Controller, and physical link interface is named as AMP_Link.

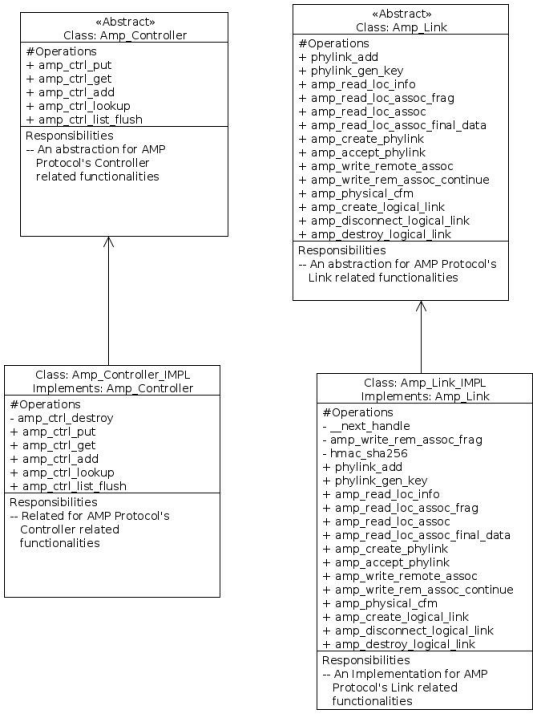


Figure 2.2: Class diagrams of AMP

2.6 Issues Faced

We faced various issues while integrating this new bluetooth driver and designing an object oriented bluetooth driver for MOOL. The issues are as follows:

- 'class', 'new' are keywords in C++ code, but in C they are not. So we need to rename them. We replaced every class with classx and new with newx.
- (void *)pointer is allowed in C ,but it is not allowed in C++, so we converted it into the specific cast.
- C language allows declaration of a structure inside the sizeof operator, whereas C++ does not. So, all the structure definition was kept outside of the sizeof operator.
- enum constants is not supported, we have replaced all enums with macros.
- __builtin_choose_expr function is a compiler level optimization, and available in C compiler, but in C++ it is not available. To get that support we need to patch the compiler, or we can replace all the __builtin_choose_expr functions with appropriate if else statement.

Part II

A MULTI-CORE OBJECT-ORIENTED SCHEDULER FOR MOOL

CHAPTER 3

Background

Linux scheduler allocates CPU to the tasks in a time sharing basis. It is called time quantum or time slice which is used to allocate slot for each process. Whenever process returns from kernel space it re-adjusts the priority of all the ready processes. Taking the feedbacks and changing the priority decision is a challenging task in Linux scheduler. This is basically called **round robin with multilevel feedback queue**. Context switching between the processes is also an overhead to the scheduler.

Memory subsystem and Linux scheduler are the important parts of a kernel. Right now Linux kernel does not work efficiently on many-core systems. Research works like [FOS] are designing a new operating system for many-core systems. Our approach is to scale Linux kernel for many-core systems by implementing new scheduler for many-core systems.

A basic scheduler algorithm is shown below:

Algorithm 1: LINUX PROCESS scheduler algorithm

```
1 while [RunQueue is not empty] do
2     moo for [All processes in RunQueue] do
3         Execute the highest priority process.
4         if No more process then
5             Sleep;
6             Interrupt: Wake up;
```

The above code is an overview of how Linux process scheduler is implemented. It takes processes from the runqueues which are loaded in the main memory and

then executes each process on a priority basis.Each core has one red-black(rb) tree which stores the processes. This is a per-core data structure. Runqueue is another per-core data structure on top of rb tree. Runqueues not only store the processes in order of their priority but also store other information like core utilization, various core related events.

The objectives of process scheduler can be summarized as follows.

- We should be fair to all the process.
- Scheduler latency is an important part of the scheduler. Its might be tricky to find an optimal point,where the scheduler latency will be less.
- We should also take care of the CPU utilization and throughput. Both throughput and utilization should be increased.
- We can also find patterns in the execution of scheduler on a set of processes and use that to assign processes to the cores. This approach works for Linux server machine where the set of processes is well-defined. Thus, we can predict server's lack of efficiency during the peak time.

We now discuss about the modern scheduler ,which is more efficient than, what we discuss till now. Linux scheduler ensures that it will take constant time to execute a process in a run queue (O(1) execution of each process). The execution time doesn't depend on the no. of processes in the runqueue. Symmetric multi processing(SMP) is a very useful feature people have looked into. Linux kernel was not designed based on SMP.SMP support is available since kernel version 2.6. Current kernel code uses all the cores using SMP scheduling technique. Mainly load balancing in the scheduler matters the most in the kernel scheduler in a SMP enables system. Batch processing in the scheduler has also been implemented. This is basically gets executed when the interactive process is less than the CPU bound process.

3.1 Scheduling Parameters

To optimize the scheduler, kernel sets the parameter depending on the current state of the CPU.This is a challenging thing in the Linux kernel. Each element in a process table has a priority field attached with it. If a process just used one CPU , it will be assigned a lower priority because of fairness. In the consequence sections we show how the kernel assigns priority dynamically when the process returns from kernel space. The priority field can be categorized into user priority,(below a threshold) and kernel priority (above a threshold value).

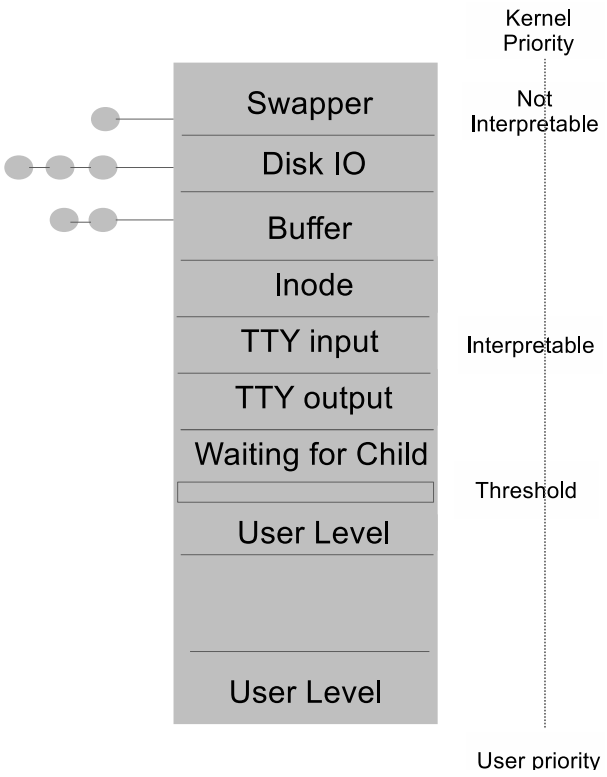


Figure 3.1: Priority queue

There are 99 priority queues available in the default Linux scheduler. There are various cases considered when assigning priority to the process. A process which is in sleep state in a low level, prevents other processes from executing. For example consider a process waiting for input/output operation(IO) and another

process waiting for a buffer. If the IO process having higher priority enters sleep state, then the buffer process won't be able to execute as the IO process holds the IO buffer with it and does not free the buffer without finishing its execution. If we allow process waiting for buffer to execute first, then the buffer will be freed soon and the second process can go ahead. If we follow this principle then no of context switch will be less, and throughout will be high. Figure 3.2 shows the process priorities in a priority queue. The circles on the lines are the processes. As discussed above that the disk IO process have higher priority than the buffer wait processes.

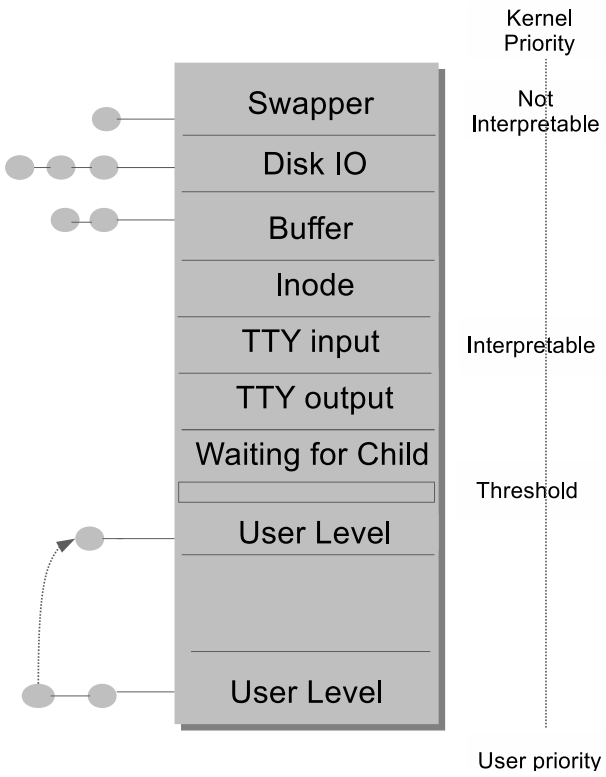


Figure 3.2: Priority management

The scheduler updates the priority when a process returns from kernel space and enters into the user-space. This is according to the policy of CFS scheduler. Some process sleeps for a long time due to low priority , where as the higher priority

processes comes and leaves the system. For instance the text editor program is a very lazy process and sleeps too much in the process queue, so those processes are high priority processes.

The policy is to change the priority, if a process is sleeps for a long time and cause starvation. Figure 3.2 shows the priority queue in user space. Kernel priorities are generally static.When kernel executes a critical region it doesn't generally changes the priority and allows the processes move to take the CPU. This is because, this will cause spending more time in the critical regions.

Considering all the facts to control the priority of the process in Linux kernel, Kernel uses the below formula

$$p = (r/c) + b + n$$

where 'p' is resultant priority. 'r' is recent CPU gain, 'c' is a constant, 'b' is the old priority, and 'n' value has a threshold within that the super user can change the priority of a process. This is basically to control the priority assigned by the users. It can't be reset by any other process. Generally child processes takes the 'n' value from the parent processes. The above scheduler code doesn't schedule the process considering that the processes come from many users. This type of scheduler policy is called process driven scheduler. The AT&T researchers developed a fairness scheme as a extension of this. This is known as **Fair share scheduler**[*Henry* 84]. It assigns CPU to a users instead of processes. We can limit the CPU usage at user level. One can define various classes of user in a system. One might assign a higher priority to the super user , whereas one might allocate less CPU time to the less privileged users. The process driven scheduler can differentiate between those. If the user has equal no of processors, and the

processes have equal priority, then each processor in a group will get the same amount of time.

We can implement this scheme. We need to divide the users into some shared group, and assign priority to each group. Then introduce a new field to the process's description called user priority or share group priority. Then update the priority generation formula discussed above accordingly.

But all of these scheduler policies might not be appropriate for a real time system. In real time system, we want quicker response ,hence we should not allow pre-emption. This is one of the reason why researcher are looking into an aspect orientated kernel, where the kernel will be looked as a service, and depending on the requirement of users we can use the appropriate kernel.[*HEM* 2012]

3.2 Linux Scheduler Design

Linux kernel is implemented in C language. It was initially implemented for single core environment. Current kernel code runs on multi core system , but they don't use all the cores in an optimal way. This is because of a poor design, as C language does not support object oriented design at language level. If we see the current scheduler design, it is noticeable that adding a new scheduler class is very difficult and time consuming. The current C scheduler has a base structure sched_class which implements the common scheduler algorithm. All the functions are registered using function pointers.

Figure 3.3 shows the basic design of C implemented Linux scheduler. We have one structure sched_class in the top. It has many functions pointers which points the common scheduler algorithms. When we call the schedule() function, we call

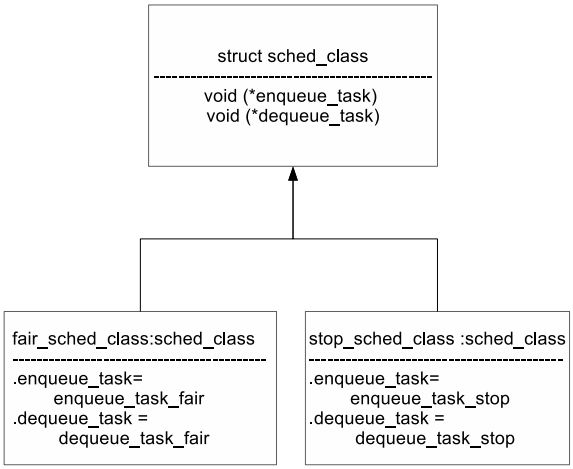


Figure 3.3: C implemented Linux scheduler Designed

using the base object sched_class. When we introduce a new scheduler policy we need to create an object of the base class and we need to register the new policy algorithm with it. For example we shown in Figure 3.3 that we have two function pointer enqueue_task and dequeue_task in sched_class struct. While applying fair scheduling policy to the system we creates fair object pointer and registers the fair scheduler algorithm(enqueue_task_fair,dequeue_task_fair). When we call the enqueue_task we call it using fair_sched_class object pointer.

3.3 Limitations of Linux Scheduler Design

In C implemented scheduler code , we see that in the base file (core.c), scheduler calls the common scheduler's functions using the base class pointer(sched_class),

and at some places it calls the fair class policy using the fair_sched_class struct object pointer. This mixture of the pointers make the scheduler code complicated. If we want to introduce a new scheduling policy it will be very difficult to find out the proper places and call the new scheduler algorithm using the new scheduler object pointer,hence it makes the code less maintainable.

Also if we want each core to execute separate scheduler policy, it might not be possible with out changing the Linux scheduler design as the current code is a mixture of all the scheduler class pointers.

3.4 MOOL: Minimalistic Object Oriented Linux

[*MOOL*] MOOL builds a wrapper and provide the developers write the device drivers in C++. An object oriented device driver is easier to maintain. Low cohesion and high coupling issue is addressed by various writing device drivers in C++. Our Linux distribution is minimal object-oriented because it builds the wrapper and integrate minimal set of object-oriented components.This wrappers basically encapsulates various operation of kernel's subsystem.The developers need to create an object of the class and call the appropriate functions. When device drivers are to be integrated from a C version to C++ version, one should check and gather the call back functions, because those functions can't be put inside a class. This is because of name mangling in C++. The kernel won't be able to identify those functions, and will cause an undefined reference error. We have to register those call back functions explicitly in the kernel and from those registered functions we can invoke the appropriate call back class functions. The procedures start by looking and differentiate the data and then encapsulate the data

in various classes. Depending on those data various relationship can be drawn. So, by separating the data and putting them in a class sometimes gives better maintainable code for the upcoming versions. Performance might get degrade after re-engineering a C driver into an object-oriented using C++. C++ drivers have a minimal performance overhead , but that is very low and we can ignore.

3.5 Related Work

We now explain some of the ongoing works in Linux scheduler. There are some works which transfers GPU call from user-space to the kernel space.

Gdev[*gdev*] provides a whole ecosystem consisting of resource management, virtualization, data swapping, etc. through APIs. Both system call and GPU call are made through the API. When we invoke CUDA functions through the Gdev API, it gets converted into an actual GPU call while system calls are handled by the operating system. This is why Gdev is known as API driven eco-system.

Timegraph[*TM*] is a command driven scheduler for GPU. It operates at device-driver level. It submits commands to control the GPU calls in the GPU channel.

[*FOS*] Factored OS is an attempt to build a scalable OS that scales over more than 1000 cores. Their research shows that spatial scheduling is more appropriate than time multiplexing in a many-core system. Spatial scheduling concentrates on where to schedule instead when to schedule.

[*GERM*] Graphics engine resource management is another command driven scheduling approach. It provides a fine-grained scheduler that follows CFS policies.

CHAPTER 4

Design and Implementation of Multi-core Object-Oriented Scheduler

4.1 Introduction

Scheduler is one of the core components of a system. A poor design of scheduler can really hurt performance.

Linux kernel is implemented in C language. It was actually designed for single core system. Though the current kernel code runs in multi-core environments, it does not use all the cores in an optimal way. The code-base of Linux scheduler is fast evolving as new functionalities are added. The scheduler code is undergoing lot of changes. Quick fixes increase coupling between the modules. In the current code, updating and adding new features is very complicated. We allow easily extensible code of kernel subsystem using C++. We provide a design where each scheduler policy is a class. The following section highlights the design approach of our object-oriented Linux scheduler.

We created a base scheduler class mool_scheduler and created many child classes of mool_scheduler like FAIR,STOP,IDLE,RT. All derived class override the required scheduler functions. We add common scheduler functions in the base class and scheduler specific functions in the child classes.The benefit of an object-oriented scheduler is as follows.

- Easily maintainable Code.

- Easily understandable Code.
- Easily extensible code.
- Adding new scheduler policies is easy.
- We allow each CPU core to execute separate scheduling policy.
- One can introduce a group scheduling policy, where each group will be assigned a distinct scheduler object,hence different scheduler policy.

4.2 Scheduling for Multi-core: Design-I

To start with, we had a working kernel for a single core system. We focused on two designs to extend it to a multi-core system. The first design we studied is to have one scheduler object that will be shared by all the cores. (refer figure 4.1). There is one base scheduler class, which implements the scheduler algorithms common to all the scheduling classes. The derived class implements the specialized scheduler class like FAIR,STOP. While initializing the scheduler class objects, we create the scheduler object of a specialized scheduler class. For example if we want to use fair class,then we create fair_sched_obj. Then we assign this fair scheduler class object to the base scheduler class object.As we have only one base class object, which is shared by all the cores, all the CPUs access the scheduler using that base scheduler object pointer. Since the fair class scheduler object is assigned to the base scheduler class object, all the cores use fair scheduling policy.

This design approach has one drawback. As the scheduler object is shared by all the cores, it has to be locked to synchronize operations across the cores. This becomes a bottleneck. In object-oriented Linux scheduler code, object state is not an issue, as we only wrap the pointers.

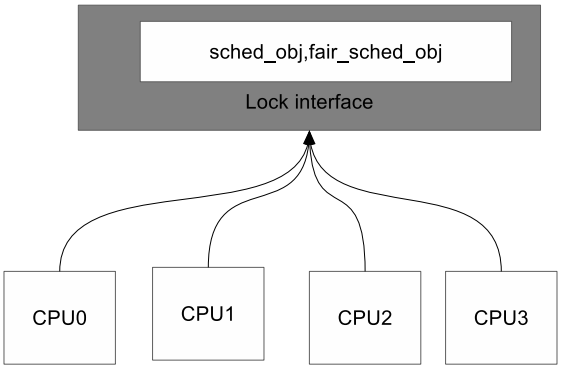


Figure 4.1: Scheduling for multi-core:Design-I

4.3 Scheduling for Multi-core: Design-II

In this section we introduce another design for multi-core Linux scheduler. This design is similar to [TSS]thread specific storage pattern. In thread specific storage pattern, a thread specific local storage is created in the global context, and each thread gets a pointer to access those variables. This reduces contention between threads. We follow a similar design in our multi-core object-oriented scheduler. Each CPU creates its own per-cpu scheduler object to access the scheduler.

Figure 4.2 shows our approach. Here all CPUs has their own base scheduler object sched_obj. When CPU calls the schedule() function, it dereference its own scheduler object pointer and calls the scheduler.

This approach has one Limitation. With this approach, the no. of per-cpu

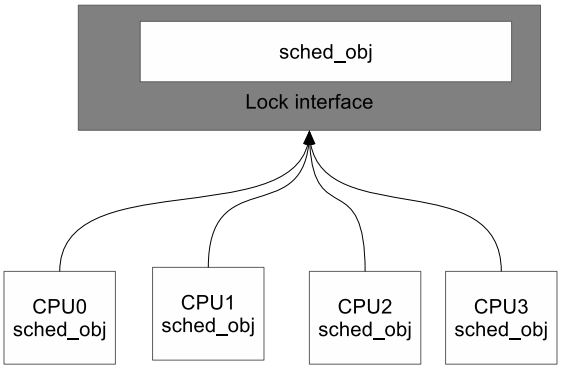


Figure 4.2: Scheduling for multi-core:Design-II

variable increases. We shouldn't use per-cpu variable unless it is highly required. Every time we access the per-cpu variable it calculates the offset value and advances the pointer. The scheduler is a core part of the system. At each steps kernel calls the schedule() function. To access the per-cpu object kernel calculates the offset value for . data.percpu sections. Every-time kernel accesses the scheduler function ,it calculates the offset value, and advances the pointer. Thus the scheduler latency might increase if there are frequent calls.

Figure 4.3 shows how CPUs get their per-cpu objects. reloc_hide macro takes the object pointer and cpu offset. Then it calculates the offset value and forwards the pointer to the appropriate CPU section.

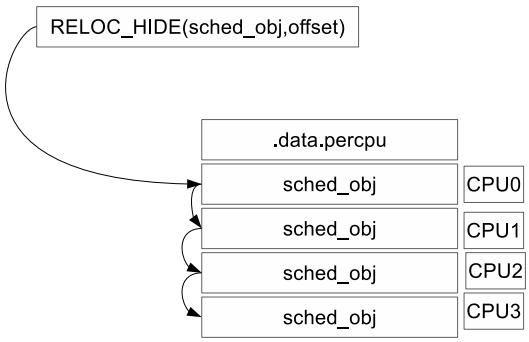


Figure 4.3: Accessing per-cpu sched_obj

4.4 Design Adopted:Design-II

We started implementing design-II approach as it reduces the contention period. We worked with 3.7 Linux kernel code that has one main file called core.c. This file defines the structure sched_class which is analogous to a base scheduler class. It contains various function pointers, which can be assigned dynamically to one of the structs like fair, stop, rt, idle, etc.

Figure 4.4 shows the design of our object oriented scheduler. The top most diagram is mool_scheduler class, which is an abstract class. Other classes like FAIR,STOP are derived from mool_scheduler. All other scheduling classes like FAIR,STOP are implemented as child classes of mool_scheduler. The base class implements all the common scheduling algorithms. The derived scheduler classes

have one object for each CPU. While initializing the scheduler, we assign the derived class pointer to the base class pointer.

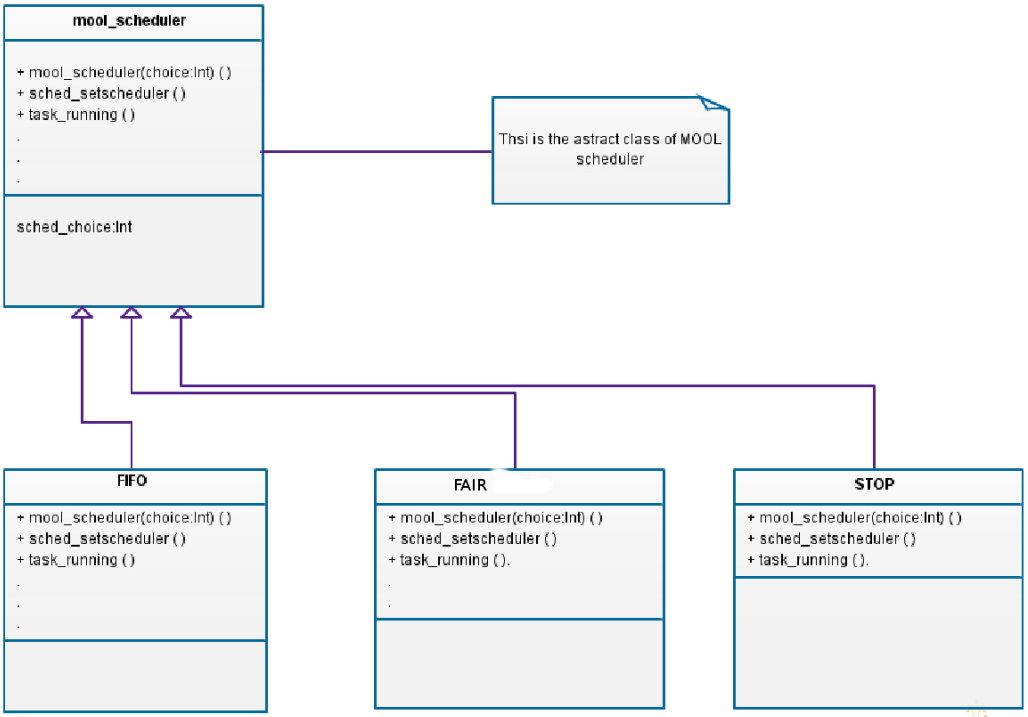


Figure 4.4: OO Scheduler Design

4.5 Implementation

It is required to register the scheduler service to each core of a system. When we switch the power button, boot monitor code starts executing. It starts booting from a predefined address code. Boot-monitor decompresses the kernel image from the memory and loads it into the main memory. Then the kernel image creates and initializes various data structures. It also creates some user processes, and boots the system.

Figure 4.5 shows the boot up process in a SMP enabled system. After decompressing the kernel image, the kernel initializes RAM, cache, MMU to CPU

0. CPU0 initializes the data structures that are common to all the cores and then boots up the other cores. Secondary CPUs boot using the same kernel code, but they start at some specific address location, so that they can simply configure their own data structures and skip all the resources initialized by CPU0.

4.6 Initializing the Scheduler

The processor independent kernel code starts from start_kernel function in main.c.main.c is located under the folder init in the kernel source directory. Here some part of the code is architecture specific. After setting up the architecture and percpu data structure, kernel initializes the Linux scheduler. sched_init function takes care of initialization of the scheduler. After calling this function in main.c ,kernel schedules processes in CPU0 only. After the initialization of scheduler, CPU0 infinitely loops and checks the TIF_NEED_RESCHED flag. If there is no more processes to be scheduled, it goes to the idle state and waits for interrupts to happen.

Whenever some process is there in the runqueue, an interrupt is raised and the CPUs wakes up and schedule the processes , and keeps checking the TIF_NEED_RESCHED flag.

4.7 Assigning Scheduler Class

As discussed in the previous section, the scheduler initialization happens in sched_init function. The proper scheduler classes are registered in the sched_init function. Fair class is the default class in modern Linux system. We assigned this default scheduler class object in sched_init function.

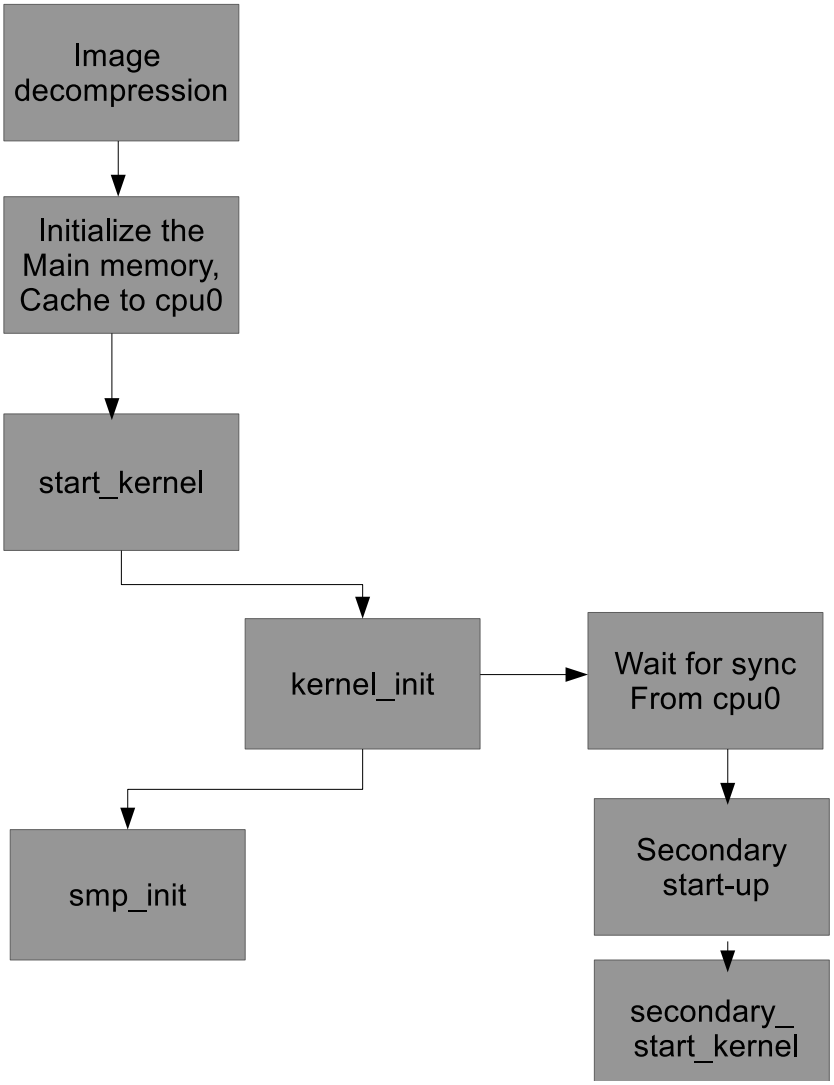


Figure 4.5: Boot-up process of Linux kernel

```
sched_init(){
.
.
for_each_possible_cpu(i){
      //initialize the rq.
}

//create the scheduler object and assign it

fair_obj=new &fair_sched_class_mool_obj()
c->sched_class =&fair_obj;

 sched_obj_ptr= &fair_obj;
.
.
}
```

Figure 4.6: scheduler initialization code

Figure 4.6 shows the code inside the sched_init function. The for loop initializes all the per-cpu data structure ,e.g. runqueues. Then it creates an object of the fair scheduler class and assigns it to he default scheduler class object. We need to pass a scheduler object pointer in task_structure objects, hence we need one separate scheduler object. Here c->sched_class holds the scheduler pointer in the task structure pointer c.

4.8 Fetching the Process from the Runqueue

The order of assigning process to CPU cores is important in Linux scheduler because it has to be fair to all the processes. Each core maintains one red-black(rb) tree pointer to store the processes. This is a per-cpu data structure. On top of this data structure, we have runqueues which is again a per-cpu variable. Runqueues not only store the processes but also maintain core utilization and CPU load informations.

The function pick_next_task(struct rq *rq) fetches the next process to be scheduled. This function is present in each scheduler class. This function iterates over all the scheduling classes and takes task from the corresponding class objects. Each scheduling class has some priority. This functions iterates the scheduling classes according to their priority.

Algorithm 2: PICK_NEXT_TASK

```
1 for cp = sched_classes; cp; cp = cp− > next do
2     p = cp− > pick_next_task(rq)
3     if p then
4         return p
5     else
6         continue
```

The above algorithm shows that *cp* - class pointer iterates over all the scheduling classes, linked through next pointer.It checks whether the task is NULL or not. If the task is not NULL then it returns the task. If no task is returned by the previous higher priority class, then the next scheduling class gets a chance.

4.9 Assigning Next Pointers

We have to also assign the next pointer in the sched_init function. But it may happen that we need to create some scheduler class after scheduler initialization in the sched_init function. In every place we need to assign the next pointer as well. Instead what we did is, we assigned the next pointer in the constructor of the scheduling class. Whenever we calls the constructor the next pointer gets assigned. The preferences of classes is stop,rt,fair,idle respectively in our current scheduler. It means stop class has the higher priority and idle has the lowest.

At some places in the C code for scheduler, they call the base class implemented functions even if they use some other scheduler class. The function pick_next_task is one of them. In the object-oriented code, we call the method statically as mool_scheduler::pick_next_task(rq);

4.10 Issues and Debugging

We faced some issues while working with the multi-core scheduler code. Debugging these issues took a lot of time. We debugged using printk and gdb. We found printk functions as unsafe. When we used printk without proper knowledge of the context, the system hung. gdb was helpful in these cases. The issues and solutions

are explained below.

4.11 Null Pointer De-reference

We got null pointer dereference error while booting the kernel. This happened after the scheduler is initialized. As CPU1 was getting a null pointer dereference error, we thought CPU1 needs a separate scheduler pointer. So we re-designed the Linux scheduler according to design-II explained earlier. The details of the issues and solutions are discussed in the following sections.

4.12 Per-CPU Scheduler Class

We created per-cpu scheduler class so that each core gets a distinct scheduler object. The following paragraph shows how we work with per-cpu data structure.

extern unsigned long __per_cpu_offset[NR_CPUS]

#define per_cpu_offset(x) (__per_cpu_offset[x])

per_cpu_offset calculate the memory space required for a per-cpu variable.

#define DEFINE_PER_CPU(type, name) __attribute__((__section__(".data.percpu")))

__typeof__(type) per_cpu__##name

This DEFINE_PER_CPU macro defines a per-cpu variable and add it to the .data.percpu section.

#define per_cpu(var, cpu) (*RELOC_HIDE(&per_cpp__##var, __per_cpu_offset[cpu]))

#define __get_cpu_var(var) per_cpu(var, smp_processor_id())

The macro RELOC_HIDE(ptr, offset) simply advances ptr by the given offset in bytes (regardless of the pointer type).

When we call DEFINE_PER_CPU(int, var), an integer __per_cpu_var gets created in .data.percpu section.This section is a special section and reserved for only per-cpu data structure. When the kernel loads, it loads the .data.per-cpu section many times. That is for each CPU,this section loads once .Each CPU loads their per_cpu variable in its own place of .data.percpu section. The __per_cpu_offset array is filled with the gap between the per-cpu variable copies. If 2000 bytes of per CPU data is used,then this macro __per_cpu_offset[n] converts it to 2000*n. The symbol per_cpu__var relocates, during load, to CPU 0's per_cpu__var. Now while using the per-cpu variable we call __get_cpu_var(var). It increments the preempt counter. This is because if we do not disable the pre-emption then the object state might be lost. So every time when we take the per-cpu variable we disable the pre-emption and after working with the variables we enable the pre-emption . When the kernel runs on say , third CPU , i.e. CPU 2, __get_cpu_var(var) translates to RELOC_HIDE(&per_cpu__var, __per_cpu_offset[2]). This start looking from CPU 0's var, and it adds the offset between CPU 0's data and CPU 2's data, and it dereferences the pointer.

In our case the variable var is a class object.In the header file of each class we declared the per-cpu variable using DECLARE_PER_CPU_SHARED_ALIGNED macro , and in the cc files we loaded the per-cpu variables using DEFINE_PER_CPU_SHARED_ALI macro. The code looks as follows.

static DECLARE_PER_CPU_SHARED_ALIGNED(class fair_sched_class_mool, fair_object);

static DEFINE_PER_CPU_SHARED_ALIGNED(class fair_sched_class_mool, fair_object);

4.13 Initializing per_cpu Objects

sched_init() function will initialize the per-cpu objects . All the scheduler related per-cpu data structure gets initialized in sched_init() function. Using the macro for_each_possible_cpu(i) we iterate over all the CPUs and get the per-cpu data structure and initialize it. We assigned a fair class object to all the CPUs. Initializing the next pointers happens in this function.

4.14 Issue: Schedule While Atomic

There are multiple ways we actually access the per-cpu object pointers. We use per_cpu macro and also we use get_cpu_var macro. get_cpu_var macro disables the pre-emption and it deferences the pointer. After using the per-cpu variable we do put_cpu_var which enables the pre-emption. When we call the schedule() function for each per-cpu scheduler class object, then we call the following code.
mool_scheduler mool_sched = get_cpu_var(sched_object);
mool_sched− >schedule();

schedule() function is a recursive function. When we disable the pre-emption and schedule a new process, the preempt_count increases recursively. This causes an error "bug while atomic". The reason is as follows: if we disable the pre-emption, the process context becomes atomic. We should not schedule a process until the current process finishes its execution. Because we are violating the scheduler atomic principle, the kernel throws the above error and exits from the flow of kernel execution.

The solution is to use per_cpu macro instead of get_cpu_var macro. This macro

does not alter the pre-emption flag. But it also does not guarantee that the object state is not lost. Currently we have very few class members in scheduler classes, other pointers like rq,task_struct are wrapped. Hence it does not create problem if we use per_cpu macro. However, whenever we need to ensure no data loss, we must use get_cpu_var macro instead of per_cpu macro.

4.15 Virtual Functions in .data.percpu Section

The whole idea is based on polymorphism; so many functions are virtual in the base class. We override those functions in the child classes. After adding this per-cpu class objects to the scheduler we noticed that all other class members except the virtual functions works properly, but for the virtual functions it is not able to call those functions properly. This problem is because of inaccessibility of vtable pointers across the cores.

4.16 Object Wrappers

To solve the null pointer on virtual functions problem, we assigned one more scheduler class pointer inside every scheduler class. Those are shown below. In the constructor of each scheduler class, we initialize the pointer as NULL. The idea here is each per-cpu object holds another pointer which is not percpu. And each core takes the inner object and it calls the appropriate functions.

mool_scheduler() {

sched_obj_ptr_in=NULL;

fair_sched_obj_ptr_in=NULL;

rt_sched_obj_ptr_in=NULL;

stop_sched_obj_ptr_in=NULL;

idle_sched_obj_ptr_in=NULL;

}

Now, while calling the scheduler object it calls get_sched_obj functions,which is written by us. This checks whether the inner object is NULL or not. If the inner object is NULL , then it creates one inner object and it returns the new object. The algorithm is given below.

Algorithm 3: GET SCHED OBJ

```
1 fair_sched_class_mool fair_sched_obj_ptr;
  fair_sched_obj_ptr = &get_cpu_var(fair_object);
  if fair_sched_obj_ptr− >fair_sched_obj_ptr_in==NULL then
2 |  fair_sched_obj_ptr− >fair_sched_obj_ptr_in=new fair_sched_class_mool();
3 put_cpu_var(fair_object); return fair_sched_obj_ptr− >fair_sched_obj_ptr_in;
```

4.17 Issues with Design-II

After making those changes, we found that design-II is not a easy choice for multi-core scheduler. We noticed that with this design kernel is not able to create the kernel objects like workqueue, kthread. This is because there is a limitation of thread specific storage[TSS]. We should not use TSS when the state of one thread passes to other threads. Here in the scheduler code, we need to pass the state of struct completion to other CPUs. This is because when we create kernel object like kthread , it fork a process. At the initial stage the process stays in sleep mode. It calls the scheduler to wake the process up. It passes a completion struct object with the process, and keeps looping and checking the state. As soon as the process

is waked up, the current CPU context comes out of the loop and proceed. In our case the state was not setting up properly,hence the current CPU was looping for a long time. The other CPU detected it as a stall and thrown the stall error.

4.18 Adopted Design: Design-I

To make our object oriented scheduler work,we shifted the code from design-II to design-I , where there was only one scheduler object. All the CPUs share that object to call the scheduler. So we removed all the per-cpu object and created only one global base scheduler object. For each derived class, we are also creating separate objects. This is because there are some places where we need to call the scheduler policy specific algorithm. There we use those pointers. For every task structure we need to pass a scheduler object also. Hence one more object pointer was created for task_struct. After doing this changes the kernel has booted up.

4.19 Evaluation

The results are evaluated in sysbench and hack-bench. Hack-bench creates fix number of processes in some time interval and records the execution time. Those processes are basically creates server and client and communicate between those.Where sysbench actually create no of thread and creates some mutexes and allows concurrent access of those mutexes to the threads, and yields the threads. It does some specified no of yields also.

The system configuration is given below.

Processor: Intel(R) Xeon(R) CPU E5-1620 0 @ 3.60GHz.

No of Core:8.

Memory Available: 8124592kB.

GPU: NVIDIA GK107[Quardo 410]

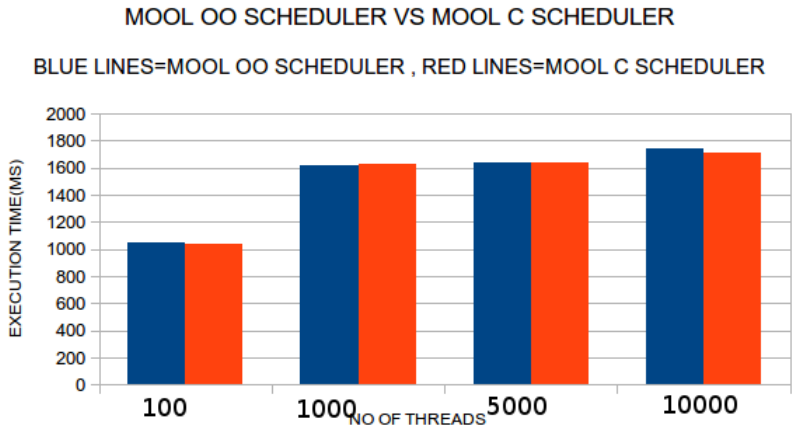


Figure 4.7: MOOL Object oriented scheduler vs MOOL C scheduler

Figure 4.7 shows the comparison between MOOL C implemented scheduler and MOOL object oriented scheduler. It shows that the average time taken by the MOOL Object oriented scheduler is 1508.74 milliseconds, whereas in MOOL C implemented scheduler the average time taken is 1501.05 milliseconds.

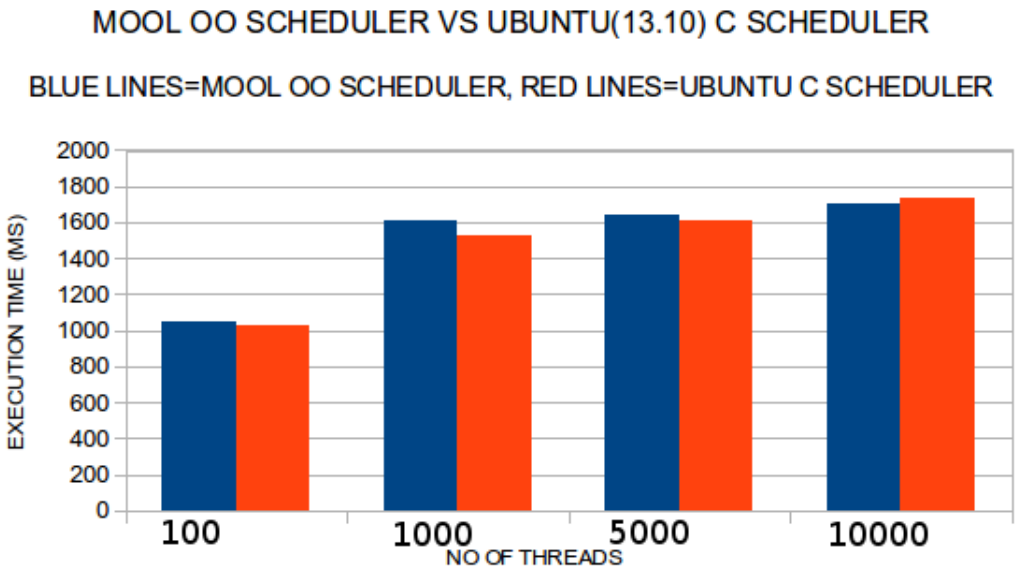


Figure 4.8: C scheduler in UBUNTU machine vs MOOL OO scheduler

Figure 4.8 shows the comparison between C implemented scheduler in an Ubuntu machine and MOOL object oriented scheduler. Here the delays are like 0.2813 sec because of c++ overhead. The average time taken by the Ubuntu

scheduler is 1473.135 milliseconds whereas MOOL OO scheduler took 1501.265 milliseconds.

Part III

INTEGRATING GPU ASSISTED SCHEDULER(GAS) IN MOOL

CHAPTER 5

Integrating GPU Assisted scheduler in MOOL

5.1 Introducing GAS

Many-core systems are becoming popular. The current Linux scheduler is not suited for Many-core systems as it doesn't take care of the topology, intra-core component utilization, etc. The system might heat up very quickly (hotspots). Also, it is difficult to include all these parameters and set them optimally as we need low latency scheduler($O(1)$ scheduling).

S J Balaji has proposed a GPU assisted scheduler (GAS) for many-core systems where the optimization problem to decide process-to-core mapping is run on the GPU. This enables low latency optimal scheduling. In this thesis, we integrate the GPU assisted scheduler into MOOL.

Current Linux scheduler is based on temporal scheduling, whereas spatial scheduling is more suitable for Many-core systems[*FOS*] . GAS provides spatial scheduling. It uses K means clustering to group the processes. GAS run-time works as a coordination master. It polls the system profiler to collect the data. Whenever the system is in imbalanced state, GAS calls the optimizer which uses GPU to improve performance.

Perf utility gathers the process information and stores it in shared memory. GAS optimizer takes these information from shared memory. The perf utility is a performance analysing tool for Linux. It has very less overhead.A red-black tree

is used to transfer data efficiently between system profiler and optimizer. GAS allows us to set various parameters to tune system performance. For balancing CPU load efficiently, we need to set K properly. We need to decide when to invoke the optimization, to get better performance.

5.2 My Contribution: Adding Kernel Module for GAS

GAS uses a stability metric to decide if the system is imbalanced. It takes the process utilization vectors and calculates the mean and the variance of the process utilizations, and calculates the stability of the system.

5.3 Communication Between User-space and Kernel-space

Kernel module can not directly communicate to a user-space program. There are multiple ways to communicate between user-space and kernel-space. Netlink is a socket based approach which has one module at the kernel, and a program at the user-space. The user-space process opens a socket and listen to a port opened by the kernel module. This socket based approach is not a reliable approach for scheduler. Because any-time the socket might break, and the kernel module disconnects from GAS.

Other way is through VFS interface. Various VFS implementation are available in Linux environment. We used proc virtual file system here. The proc file system API has two call back function, one is procfile_read , another one is procfile_write. To check the stability value,the user-space routine calls procfile_read through VFS

API. It then writes the stability value using procfile_write function. The procfile_write function calls the stability calculator. After calculating the stability value it writes it into a proc file. The user-space routine reads the stability value and calls GAS.

5.4 Issues Faced

There are various issues we faced while integrating GAS in MOOL. The below sections explain these issues in detail.

5.4.1 Perf Patches

GAS uses perf utility to collect various process events. There was a version mismatch of perf utility in MOOl. So we got segmentation fault while using the default perf utility in MOOL. Then we applied a patch for perf utility in MOOL. After applying the patch it executed properly.

5.4.2 Null Pointer De-reference

The stability calculator actually allocates memory for storing the process data.Memory allocation in the kernel space is not simple like in the user-space. We used kmalloc for allocating the memory. One can use vmalloc also to allocate memory in the kernel-space. The difference between the vmalloc and kmalloc is explained below.

5.4.3 Vmalloc and Kmalloc

Kmalloc allows to create contiguous memory in RAM, where Vmalloc does not guarantee that. But Vmalloc allocates contiguous memory pages in the kernel memory block. To allocate small size memory ,we use Kmalloc.We do not use Kmalloc for allocating a big memory chunk,as it might be difficult to allocate a big contiguous memory in RAM. The file slab.h defines Kmalloc functions in the kernel space.Kmalloc performance depends on the second argument it takes. It is a kernel flag to control the memory allocation through Kmalloc. These popular flags are explained below.

$void * kmalloc(size\_tsize, int flags)$

- GFP_ATOMIC(This flag indicates that the process can never sleep before allocating the memory, generally this is used in various interrupt handlers and code outside of process context).
- GFP_KERNEL (This is normal allocation of kernel memory where the process may or may not sleep.)
- GFP_USER(This is generally used for user-space pages. The process may sleep.)

As explained we need to choose the correct flag. We tried with GFP_KERNEL first, but that was not stable. It was not able to allocate the memory all the time.Then we used GFP_ATOMIC flag. But when we called the kernel module again again the kernel crashed because of continuous memory allocation and reallocation. Then we shifted the memory allocation part in the global section. We initialized the memory only once while the module loads. After that we reset the memory while calling the stability calculator.

5.5 Evaluation

We evaluated GAS using hack-bench workload. Hack-bench creates the processes, and record the process completion time. The processes are client server program. They communicate using sockets. We show the system stability after integrating GAS in Linux kernel.

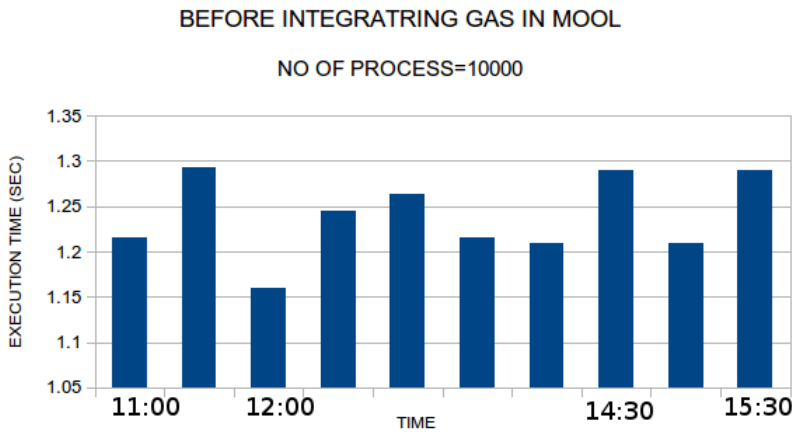


Figure 5.1: OO scheduler before integrating GAS in MOOL

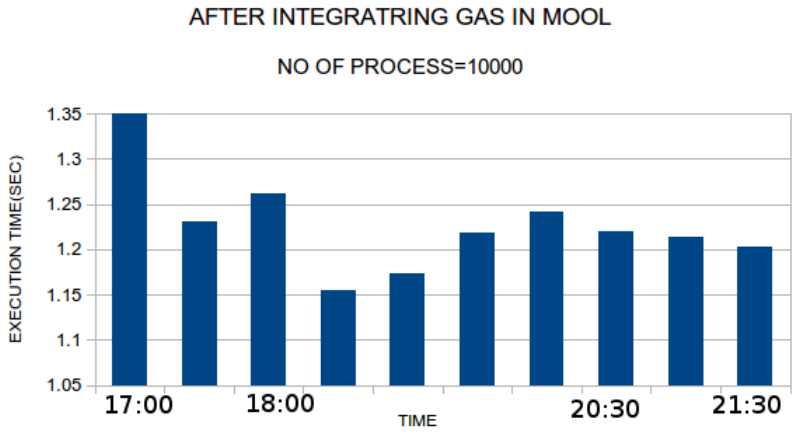


Figure 5.2: OO scheduler after integrating GAS in MOOL

Figure 5.1 shows the system stability before integrating GAS in MOOL OO scheduler and the figure 5.2 shows the system stability after integrating GAS in MOOL OO scheduler. The average time taken after integrating GAS in MOOL is

1.213 sec, which is less than the time taken before integrating GAS (1.2414444 sec).

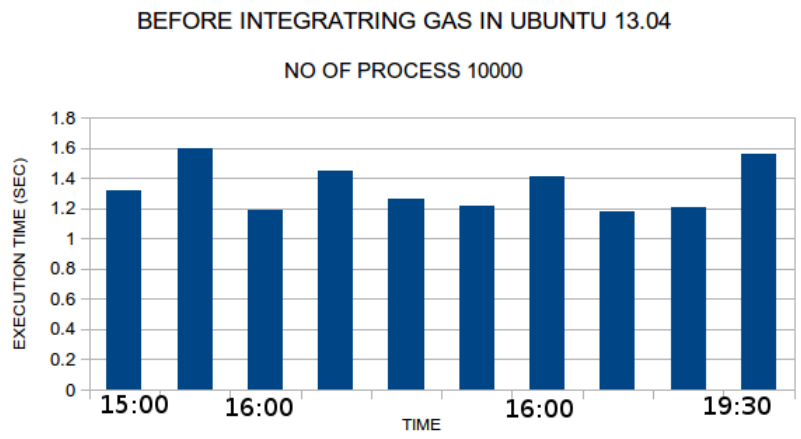


Figure 5.3: C scheduler before integrating GAS in UBUNTU

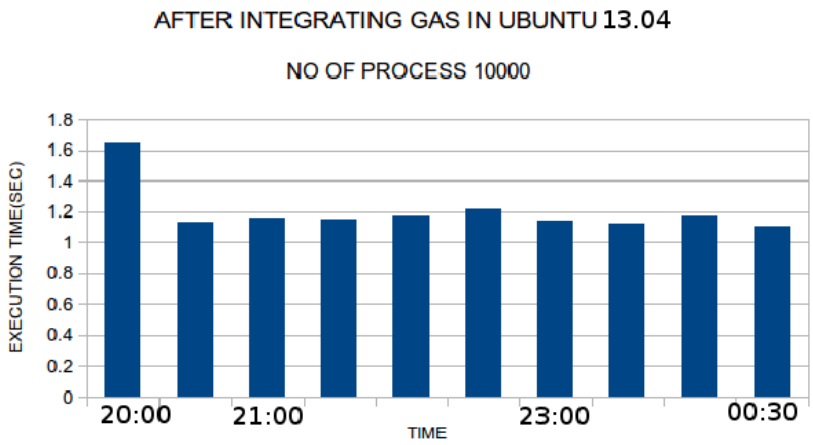


Figure 5.4: C scheduler after integrating GAS in UBUNTU

Figure 5.3 shows the system stability before integrating GAS in Ubuntu machine, whereas Figure 5.4 shows the system stability in UBUNTU machine.

REFERENCES

[1] [HEM] : *Dhara: A Service Abstraction-Based OS Kernel Design Model. IEEE 17th International Conference on Engineering of Complex Computer Systems, Paris, France France July 18-July 20,2012*
Dharanipragada Janakiram Hemang Mehta S.J. Balaji.

[2] [HENRY 84] *:The UNIX System: The Fair Share Scheduler, G. J. Henry*

[3] [Gdev] *:Gdev: First-Class GPU Resource Management in the Operating System,Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott B randt,Department of Computer Science, UC Santa Cruz.*

[4] [TM] *:TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Env ironments,Shinpei Kato,Karthik Lakshmananand Ragunathan (Raj) Rajkumar,Yutaka Ishikawa,Department of Electrical and Computer Engineering, Carnegie Mellon University,Department of Computer Science, The University of Tokyo*

[5] [FOS] *:Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores, David Wentzlaff and Anant Agarwal Computer Science and Artificial Intelligence Laboratory Massachusetts Institute of Technology Cambridge, MA 02139wentzlaf, agarwal,@csail.mit.edu*

[6] [GERM] *Graphic Engine Resource Management,Mikhail Bautin Ashok Dwarakinath Tzi-cker Chiueh,Computer Science Department, Stony Brook Universitymbautin, ashok,chiueh@cs.sunysb.edu*

[7] [MOOl] *Object-oriented wrappers for the Linux kernel,D. Janakiram, Ashok Gunnam*,N. Suneetha,Vineet Rajani,K. Vinay Kumar Reddy*

[8] [TSS] *Thread-Specific Storage for C/C++ An Object Behavioral Pattern for Accessing per-Thread State Efficiently by Douglas C. Schmidt, Nat Pryce,Timothy H. Harrison*