

Santa Clara University
DEPARTMENT of COMPUTER ENGINEERING

Date: June 12, 2007

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY
SUPERVISION BY

Kuan Ju Ho

ENTITLED

Workload-Adaptative Memory Scheduler

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF

BACHELOR OF SCIENCE IN COMPUTER ENGINEERING

THESIS ADVISOR

DEPARTMENTCHAIR

Workload-Adaptative Memory Scheduler

by

Kuan Ju Ho

SENIOR DESIGN PROJECT REPORT

Submitted in partial fulfillment of the requirements
for the degree of
Bachelor of Science in Computer Engineering
School of Engineering
Santa Clara University

Santa Clara, California

June 12, 2007

Table of Contents

	<u>Page</u>
Abstract.....	4
Acknowledgements.....	5
Chapter 1 – Introduction.....	6
Chapter 2 – The Project.....	7
Chapter 2.1 – Understand the process scheduler.....	7
Chapter 2.2 – Modify the scheduler.....	8
Chapter 2.3 – Test the modified scheduler.....	10
Chapter 2.4 – Understand the memory manager.....	10
Chapter 2.5 – Modify the memory manager.....	11
Chapter 2.6 – Test the modified memory manager.....	12
Chapter 3 – Issues.....	13
Chapter 3.1 – Ethical.....	13
Chapter 3.2 – Social.....	13
Chapter 3.3 – Political.....	13
Chapter 3.4 – Economic.....	13
Chapter 3.5 – Health and Safety.....	13
Chapter 3.6 – Manufacturability.....	14
Chapter 3.7 – Sustainability.....	14
Chapter 3.8 – Environmental Impact.....	14
Chapter 3.9 – Usability.....	14
Chapter 3.10 – Lifelong Learning.....	14
Chapter 3.11 – Compassion.....	14
Chapter 4 – Conclusion.....	15
Chapter 5 – References.....	16
Appendix A.....	17
Appendix B.....	19
Appendix C.....	26

Abstract

The need for concurrency has driven the development of several different scheduling algorithms. Said algorithms come in all sorts of different flavors nowadays. One of them was created with the idea of providing computing power as a profit-based business. The fact that some customers are willing to pay more than others in order to acquire more computing power means that the scheduling algorithm must be able to accept input parameters such as the percentage of CPU time that each process is supposed to get. This is later reflected in the amount of time that the process actually spends on the CPU doing useful work. Several algorithms to achieve this have been developed by several different companies.

However, my focus is not on the scheduler itself but on memory. For most programmers, memory has usually been a commodity and, in few occasions, we find ourselves in dire need of memory. Should such an occasion occur, though, we will find that our current memory manager will struggle to find the optimal page to be swapped out in an environment where each process needs to meet a certain percentage of CPU time. Therefore, the idea is to build a custom memory manager that will work alongside the scheduler to maximize the possibility of each process meeting its required CPU share. In this paper the design of a simple custom scheduler and memory manager are explained, but the results have not yet been obtained due to the difficulty in obtaining test cases large enough to cause any noticeable lack of memory, as well as other timing constraints.

Keywords: scheduler, memory manager, fair share, dynamic memory allocation.

Acknowledgements

I would like to thank my advisor, Dr Silvia Figueira, without whom this project would never have come to be. Beyond the expected technical guidance, she has been an important player in making this project happen, and has supported me thoroughly during times of crisis for the project.

I also owe great thanks to the technical guidance offered by the many people that dedicate themselves to the ongoing project of the linux kernel, without which completion of the project would have been indefinitely pushed out due to the enormous complexity that the project entails.

Chapter 1 – Introduction

The development of automated algorithms to control the scheduling related to profit-based system resource sharing has been fundamentally geared towards process scheduling. Little attention has been put to the fact that memory-intensive applications will spend many CPU cycles page faulting in systems where memory provided to applications is limited.

HP's Tycoon [2] is an example of a market-based system for managing computer resources in distributed clusters in which consumers pay providers for access to a certain fraction of the available resources. This fraction share is proportional to the amount of money that the consumer is willing to provide.

Our objective is to implement a custom memory-manager that will cater for the memory requests of the different applications in such a way as to aid them in achieving their requested CPU shares. This is to be done through a dynamic allocation mechanism which will gradually converge into each process coming as close to achieving their share as allowed by the underlying hardware. This report is organized as follows: Chapter 2 describes the project, Chapter 3 discusses important issues related to the project, Chapter 4 concludes, and Chapter 5 provides references. The Appendices provide changes made to the system and code developed for testing the changes.

Chapter 2 – The Project

As explained in the introduction, a profit-based system for resource sharing consists of having the applications that pay for cycles have higher priority, whereas those that do not pay will use the CPU whenever possible. Further, we will let those applications that pay more have higher priority than those applications that pay less.

This project has been divided into the following steps:

2.1 Understand the process scheduler

2.2 Modify the scheduler

2.3 Test the modified scheduler

2.4 Understand the memory manager

2.5 Modify the memory manager

2.6 Test the modified memory manager

2.1 Understand the process scheduler

Linux scheduling is based on the time-sharing technique, in which several processes are allowed to run “concurrently,” which means that the CPU time is roughly divided into “slices,” one for each runnable process. If a currently running process is not terminated when its quantum expires, a process switch may take place. The scheduling policy is based on ranking processes according to their priority. Each process is associated with a value that denotes how likely it is to be assigned the CPU, and the process with the highest value is selected to run next. The Linux kernel differentiates two types of processes: real-time processes, which have a static priority assigned to them by the user, and non-real-time processes, which have a dynamic priority that is adjusted periodically by the scheduler based on the process’ base time quantum and the number of ticks of CPU time left to the process before its quantum expires. The fact that real-time processes have a static priority that is never changed by the scheduler, added to the fact that the static priorities of real-time processes is always higher than the dynamic priorities of conventional processes, we treated our applications as real-time processes. We found that, by setting the priority of our process to real-time, we are guaranteed that the scheduler

will start running conventional processes only when our process have left the TASK_RUNNING state [1], i.e., when none of the applications paying for CPU cycles needs the CPU.

2.2 Modify the scheduler

For the sake of simplicity of implementation, a system call was introduced whereby each process can set the ratio of CPU cycles that it is supposed to get, as well as its corresponding CPU percentage. This can be easily changed to a system call accepting a structure that maps process IDs to their corresponding percentages or ratios. This, however, would impose the overhead of having to loop through all the processes trying to find the ones that are included in the map. As this was an unnecessary overhead for testing purposes, it was avoided to enhance the performance of the test program as well as simplify the implementation. Since the process descriptor contains all the process-specific data, it was chosen to store each process' intended CPU ratio/percentage. Therefore, variables were added to the process descriptor as described in the section that follows:

2.2.1 The Process Descriptor

The process descriptor is defined in `/include/linux/sched.h` as a structure called `task_struct`.

For the purposes of our custom scheduler, four integer variables and one Boolean variable were added to the process descriptor:

- `unsigned int time_slice_count`: this variable contains the amount of time slices that the process has left before it needs to be switched out, and is decremented by the scheduler each time a process uses up a quantum.
- `unsigned int time_slice_count_orig`: this variable contains the ratio of time slices that the process is supposed to get, and is set by the user.
- `unsigned int time_slice_percentage`: this variable contains the percentage of time slices that the process is supposed to get, and is set by the user.

- unsigned int `time_slices_used`: this variable contains the amount of time slices that the process has already used up to the present instant in time, and is set by the scheduler.
- unsigned int `is_meeting_cpu_percentage`: this Boolean variable indicates whether the process represented by the descriptor is getting the amount of CPU that it is supposed to get, and is set by the scheduler.

After having introduced all the aforementioned variables and tested them, the scheduler needed to be modified in order to meet the requirements of our custom scheduler. To this end, two tasks were modified: the `scheduler_tick` task and the `schedule` task. These changes are explained in the sections that follow.

2.2.2 The scheduler_tick task

The `scheduler_tick` task is defined in `/kernel/sched.c` and is responsible for keeping track of the amount of CPU that each process is using. Each process has a `time_slice` variable that keeps track of the amount of scheduler ticks that the process has left before its quantum is over. Therefore, this task must be invoked at regular intervals to decrease this number. When this number reaches zero, the scheduler comes in and picks a new task to run. In our custom scheduler, however, the process should be swapped out only after it has used a number of quanta equal to the ratio that has been provided by the user. Therefore, each time the `time_slice` variable reaches zero, the variable `time_slice_count` is decremented. Only when `time_slice_count` reaches zero is the scheduler supposed to select a new task to run. The `time_slice_count` is then restored to its original value stored in `time_slice_count_orig`. Otherwise, the amount in `time_slices_used` is incremented and the process keeps running.

2.2.3 The schedule task

The `schedule` task is defined in `/kernel/sched.c` and is responsible for selecting a new task to run each time the current process has used up its time quantum. Each time the scheduler is invoked, we need to calculate whether the current process taking up the CPU is meeting its target CPU percentage. This is done by calculating the percentage of the

total CPU cycles elapsed that have been allocated to the current process. This is then transformed into a percentage and compared against the value of `time_slice_percentage` that has been set for this process by the user. A margin of error of 5% is allowed in our current implementation. A new auxiliary task has been created to take care of all the computation, and is called `is_meeting_cpu_percentage`.

The mechanism described above will work provided that the user can set the aforementioned values of `time_slice_count_orig` and `time_slice_percentage` for each process. As mentioned before, each process sets its own ratio and percentage through a system call, which for the purposes of this project has been called `sys_mysevice`, which can easily be changed to a more descriptive name. The function of this system call is simply to set the aforementioned fields in the corresponding process descriptor.

The complete source code for all scheduler changes is listed in Appendix A.

2.3 Test the modified scheduler

The modified scheduler was put to test under two different scenarios: one with two processes where the ratio of percentages is high, and one with four processes where the ratio of percentages is relatively low. Both the test programs and their respective outputs are shown in Appendix B.

2.4 Understand the memory manager

Each process has its own Page Global Directory (PGD), which is a physical page frame containing an array of `pgd_t`. Each active entry in the PGD table points to a page frame containing an array of Page Middle Directory (PMD) entries of type `pmd_t` which in turn point to page frames containing Page Table Entries (PTE) of type `pte_t`, which finally point to page frames containing the actual user data. Any given linear address may be broken up into parts to yield offsets within these three page table levels and an offset within the actual page.

Macros are defined in `asm/pgtable.h` which are important for the navigation and examination of page table entries. To navigate the page directories, three macros are provided which break up a linear address space into its component parts. `pgd_offset()` takes an address and the `mm_struct` for the process and returns the PGD entry that covers the requested address. `pmd_offset()` takes a PGD entry and an address and returns the relevant PMD. `pte_offset()` takes a PMD and returns the relevant PTE. The remainder of the linear address provided is the offset within the page. A second round of macros determine if the page table entries are present or may be used.

- `pte_none()`, `pmd_none()` and `pgd_none()` return 1 if the corresponding entry does not exist;
- `pte_present()`, `pmd_present()` and `pgd_present()` return 1 if the corresponding page table entries have the PRESENT bit set;
- `pte_clear()`, `pmd_clear()` and `pgd_clear()` will clear the corresponding page table entry;
- `pmd_bad()` and `pgd_bad()` are used to check entries when passed as input parameters to functions that may change the value of the entries. [3]

2.5 Modify the memory manager

Since the kernel will grant virtually every request for memory, as long as it has enough memory to satisfy the request, we only need to take action when a swap-out is needed, i.e., when space is not available. When a swap-out does occur, we need to guarantee that pages belonging to processes that are not meeting their respective percentages are kept in memory. If a process is not meeting its percentage, it may be due to the fact that it is page faulting a lot and thus wasting some of its cycles to read its pages in. Therefore, when a process is not meeting its percentage we instruct the kernel not to take its frames away unless absolutely necessary. To this end, since each process contains information in its process descriptor about whether it is meeting its target CPU percentage, we need only to find out to which process each frame being swapped belongs, and then check whether that process is meeting its target share. This logic is to be inserted in the task that scans the different frames in physical memory to decide which ones can be swapped out. This task is the `refill_inactive_zone` task defined in `/mm/vmscan.c`, in which we scan the

processes sequentially and determine whether the page we are trying to free belongs to a process which can let go a page. This imposes a huge overhead in execution, but we found no other way of finding out which process owns the page, as the kernel does not keep ownership information in the page structure. Changing the memory management unit to keep track of that would enable a more efficient approach.

The complete source code for all memory manager changes is listed in Appendix C.

2.6 Test the modified memory manager

As mentioned before, due to the difficulty in obtaining test cases large enough to cause any noticeable lack of memory, as well as other timing constraints, we have not yet been able to devise a way to test the modified memory manager. The kernel has been running smoothly even with a lot of applications running concurrently, but the memory manager has not been tested thoroughly.

3 Issues

3.1 Ethical:

The only ethical issue with the kernel modification is the potential to be used maliciously in a denial of service attack. This can be solved by enhancing the security of the method of introducing the mapping of processes to CPU ratios.

3.2 Social:

There are no social issues related to the modification of the kernel for commercial use.

3.3 Political:

There are no political issues related to the modification of the kernel for commercial use.

3.4 Economic:

The modification of the kernel is intended to be used in commercial applications, and therefore service providers will be able to charge customers for the services provided. This will be left to be dealt with by the provider and the customer. As for the modification itself, it has been done on the Linux kernel of Fedora Core 4, which is distributed freely. The technologies used in the project are distributable under free software licenses. The machine on which the modification was performed was borrowed from the Santa Clara University's Parallel Computing Lab, and was returned upon the completion of the project.

3.5 Health and safety:

Health and safety issues are those associated with using the computer for long periods of time.

3.6 Manufacturability:

As with every piece of software, the manufacturability of the product consists in the ability of obtaining either the original source code or a compiled executable of it. All the

source code introduced into a stock Linux kernel is shown in this document (Appendices A-C).

3.7 Sustainability:

As the Linux kernel continues to be developed, the piece of software needs to be kept up to date with the current version of the kernel.

3.8 Environmental Impact:

The environmental impact of the product is the same as that of using a computer for long periods of time.

3.9 Usability:

The modified Linux kernel poses no extra challenges to the usability of a stock Linux kernel. The software's interface to the user has been made as straightforward as possible, barring any security concerns.

3.10 Lifelong learning:

As I started out this project, I was a little overwhelmed by the size of the development project. Having completed this project, I feel I have learned a great deal about the internal workings of the kernel, and look forward to learn more should other projects offer that possibility.

3.11 Compassion:

There are no compassion issues related to the modification of the kernel for commercial use.

4 Conclusion

Before taking part of this project I had been only a Linux user. I was familiar with shell scripting, but had never tried to recompile the kernel. Throughout this project I learned various skills such as finding my way through large chunks of code and following a compilation method set by others. These skills will prove invaluable as I look forward to entering the workforce of a company that has established code-bases and methods of making things work. I have also had the privilege of learning “the hard way” when I crashed the kernel with no backup and had to re-install the whole operating system thus obliterating all the previous work that had been done. I believe that all these skills are crucial to my formation as an engineer and prepare myself to develop software in the real world.

Future work on this project may be done in providing a secure interface for the system call. A thorough testing of the memory manager might also be helpful. Another possibility is to find a more efficient way in which the memory manager can identify to which process a page belongs.

5 References

1. Bovet, Daniel P. and Cesati, Marco. Understanding the Linux kernel. October 2000. O'Reilly.
2. Tycoon. <http://tycoon.hpl.hp.com>. May 2007
3. Gorman, Mel. Understanding the Linux Virtual Memory Manager. February 2004. <http://www.csn.ul.ie/~mel/projects/vm/guide/pdf/understand.pdf>.

Appendix A - scheduler source code changes

```
/include/linux/include/sched.h (modified struct task_struct)
468 unsigned int time_slice_count;
469 unsigned int time_slice_count_orig;
470 unsigned int time_slice_percentage;
471 unsigned int time_slices_used;
472 unsigned int is_meeting_cpu_percentage;

/kernel/sched.c (modified scheduler_tick() function)
2442     if((p->policy == SCHED_RR) && !p->time_slice) {
2443         p->time_slices_used++;
2444         total_time_slices++;
2445
2446         if(!p->time_slice_count) {
2447             p->time_slice_count = p->time_slice_count_orig;
2448             set_tsk_need_resched(p);
2449
2450             /* put it at the end of the queue; */
2451             dequeue_task(p, rq->active);
2452             enqueue_task(p, rq->active);
2453         }
2454
2455         p->time_slice = task_timeslice(p);
2456         p->first_time_slice = 0;
2457     }
. . .
2635 void is_meeting_cpu_percentage(void)
2636 {
2637     struct task_struct *p;
2638
2639     for_each_process(p) {
2640         if(p->policy == SCHED_RR) {
2641             int percentage = -1;
2642
2643             if(total_time_slices != 0)
2644                 percentage = (p->time_slices_used * 100) /
total_time_slices;
```

```
2645
2646         if(percentage != -1)
2647             if(percentage > p->time_slice_percentage -
5)
2648                 p->is_meeting_cpu_percentage = 1;
2649             else
2650                 p->is_meeting_cpu_percentage = 0;
2651         }
2652     }
2653 }
. . .
(modified schedule() function)
2668     is_meeting_cpu_percentage();
. . .
```

Appendix B - scheduler test programs and outputs

myservice-user.h:

```
#include <linux/unistd.h>

_syscall2(void, myservice, int, ratio, int, percentage);
```

test1.c:

```
#include <myservice-user.h>

int main()
{
    int priority = 2;
    unsigned int i = 0;
    int pid;

    /* set the scheduling mode to real time round robin */
    syscall(__NR_sched_setscheduler, 0, 2, &priority);

    pid = fork();

    if(pid == 0) { // Process 1: 10%
        myservice(1, 10);
        for(i = 0; i < 300000000; i++)
            if((i % 10000000) == 0)
                printf("Process 1 reached %d\n", i);
        printf("Process 1 is done\n");
    } else { // Process 2: 90%
        myservice(9, 90);
        for(i = 0; i < 300000000; i++)
            if((i % 10000000) == 0)
                printf("Process 2 reached %d\n", i);
        printf("Process 2 is done\n");
    }
}
```

Output :

```
Process 1 reached 0
Process 2 reached 0
Process 2 reached 10000000
Process 2 reached 20000000
Process 2 reached 30000000
Process 2 reached 40000000
Process 2 reached 50000000
Process 2 reached 60000000
Process 2 reached 70000000
Process 2 reached 80000000
Process 2 reached 90000000
Process 1 reached 10000000
Process 2 reached 100000000
Process 2 reached 110000000
Process 2 reached 120000000
Process 2 reached 130000000
Process 2 reached 140000000
Process 2 reached 150000000
Process 2 reached 160000000
Process 2 reached 170000000
Process 2 reached 180000000
Process 1 reached 20000000
Process 2 reached 190000000
Process 2 reached 200000000
Process 2 reached 210000000
Process 2 reached 220000000
Process 2 reached 230000000
Process 2 reached 240000000
Process 2 reached 250000000
Process 2 reached 260000000
Process 2 reached 270000000
Process 1 reached 30000000
Process 2 reached 280000000
Process 2 reached 290000000
Process 2 is done
Process 1 reached 40000000
Process 1 reached 50000000
```

```
Process 1 reached 60000000
Process 1 reached 70000000
Process 1 reached 80000000
Process 1 reached 90000000
Process 1 reached 100000000
Process 1 reached 110000000
Process 1 reached 120000000
Process 1 reached 130000000
Process 1 reached 140000000
Process 1 reached 150000000
Process 1 reached 160000000
Process 1 reached 170000000
Process 1 reached 180000000
Process 1 reached 190000000
Process 1 reached 200000000
Process 1 reached 210000000
Process 1 reached 220000000
Process 1 reached 230000000
Process 1 reached 240000000
Process 1 reached 250000000
Process 1 reached 260000000
Process 1 reached 270000000
Process 1 reached 280000000
Process 1 reached 290000000
Process 1 is done
```

```
test2.c:
#include <myservice-user.h>

int main()
{
    int priority = 2;
    unsigned int i = 0;
    int pid1, pid2, pid3;

    /* set the scheduling mode to real time round robin */
    syscall(__NR_sched_setscheduler, 0, 2, &priority);
```

```

pid1 = fork();

if(pid1 == 0) {
    pid2 = fork();

    if(pid2 == 0) { // Process 1: 10%
        myservice(1, 10);
        for(i = 0; i < 300000000; i++)
            if((i % 10000000) == 0)
                printf("Process 1 reached %d\n", i);
        printf("Process 1 is done\n");
    } else { // Process 3: 30%
        myservice(3, 30);
        for(i = 0; i < 300000000; i++)
            if((i % 10000000) == 0)
                printf("Process 3 reached %d\n", i);
        printf("Process 3 is done\n");
    } else { // Process 2: 90%
        pid3 = fork();

        if(pid3 == 0) { // Process 2: 20%
            myservice(2, 20);
            for(i = 0; i < 300000000; i++)
                if((i % 10000000) == 0)
                    printf("Process 2 reached %d\n", i);
            printf("Process 2 is done\n");
        } else { // Process 4: 40%
            myservice(4, 40);
            for(i = 0; i < 300000000; i++)
                if((i % 10000000) == 0)
                    printf("Process 4 reached %d\n", i);
            printf("Process 4 is done\n");
        }
    }
}
}

```

Output :

```
Process 1 reached 0
Process 3 reached 0
Process 3 reached 10000000
Process 2 reached 0
Process 4 reached 0
Process 4 reached 10000000
Process 3 reached 20000000
Process 2 reached 10000000
Process 4 reached 20000000
Process 4 reached 30000000
Process 1 reached 10000000
Process 3 reached 30000000
Process 3 reached 40000000
Process 2 reached 20000000
Process 4 reached 40000000
Process 4 reached 50000000
Process 3 reached 50000000
Process 2 reached 30000000
Process 4 reached 60000000
Process 4 reached 70000000
Process 1 reached 20000000
Process 3 reached 60000000
Process 3 reached 70000000
Process 2 reached 40000000
Process 4 reached 80000000
Process 4 reached 90000000
Process 3 reached 80000000
Process 2 reached 50000000
Process 4 reached 100000000
Process 4 reached 110000000
Process 1 reached 30000000
Process 3 reached 90000000
Process 3 reached 100000000
Process 2 reached 60000000
Process 4 reached 120000000
Process 4 reached 130000000
Process 3 reached 110000000
```

Process 2 reached 70000000
Process 4 reached 140000000
Process 4 reached 150000000
Process 1 reached 40000000
Process 3 reached 120000000
Process 3 reached 130000000
Process 2 reached 80000000
Process 4 reached 160000000
Process 4 reached 170000000
Process 3 reached 140000000
Process 2 reached 90000000
Process 4 reached 180000000
Process 4 reached 190000000
Process 4 is done
Process 1 reached 50000000
Process 3 reached 160000000
Process 3 reached 170000000
Process 2 reached 110000000
Process 3 reached 180000000
Process 2 reached 120000000
Process 1 reached 60000000
Process 3 reached 190000000
Process 3 is done
Process 2 reached 130000000
Process 2 reached 140000000
Process 1 reached 70000000
Process 2 reached 150000000
Process 2 reached 160000000
Process 1 reached 80000000
Process 2 reached 170000000
Process 2 reached 180000000
Process 1 reached 90000000
Process 2 reached 190000000
Process 2 is done
Process 1 reached 100000000
Process 1 reached 110000000
Process 1 reached 120000000
Process 1 reached 130000000

```
Process 1 reached 140000000  
Process 1 reached 150000000  
Process 1 reached 160000000  
Process 1 reached 170000000  
Process 1 reached 180000000  
Process 1 reached 190000000  
Process 1 is done
```

Appendix C - memory manager source code changes

```
/mm/vmscan.c (modified refill_inactive_zone() function)
655  unsigned long address;
656  struct task_struct *p;
. . .
731      address = (unsigned long) page_address(page);
732
733      for_each_process(p) {
734          struct mm_struct *mm;
735          pgd_t *pgd;
736
737          mm = p->mm;
738
739          pgd = pgd_offset(mm, address);
740          if(pgd_none(*pgd) || unlikely(pgd_bad(*pgd)))
741              continue;
742          if(p->is_meeting_cpu_percentage) {
743              list_add(&page->lru, &l_inactive);
744              break ;
745          }
746      }
```