Acknowledgements

I express my deep gratitude and thank **Dr. T. Matthew Jacob** for his unbelievable patience and excellent guidance through out my thesis work. Without his moral support and gentle prodding this work wouldn't have been completed. His totally positive and well organised approach to life is worth emulating.

I thank the faculty of the department with whom I interacted during my stay here, Chairman Prof. Viswanadham, Prof. Vijay Chandru, Dr. Jayant Haritsa, Dr. Swami Manohar, Prof. Narahari for his greetings with a smiling face, Dr. Sriram for his enthusiasm in running the Computer Architecture seminar series, and Prof. Sathiya Keerthi for the concern he showed and for allowing me to use the Robotics Lab's SS20. I thank all the CSA office staff particularly Mrs. Lalitha, Mrs. Meenakshi, Mr. Achar and Mr. Mohan, who were ever willing to help.

I thank my ex-roommate Yuva for bearing with me for long, despite his personal preference of a single room and for the discussions we had. I thank Maggi, Ganju, Dixon, Aunty and Sriram for the on and off advise to work and complete fast. Thanks to Arav for opting to show the carrot (which is my own money!) available to me, upon thesis submission. Ravi, for trying his level best in provoking and angering me into completing this work. Amidst all the dust generated due to the delayed submission, there was at-least one person apart from my guide who always managed to push me towards the completion of this work. I thank KVS for his patient hearing out of my problems, the invaluable help he rendered in debugging parts of my code and for letting me free to use the system as long, and as and when, I wished to.

I also thank raja, gomu, gsri, malay, dinesh, amit, argya for the time spent with them

and nasa for helping me out. I thank SERC for the excellent facilities provided, particularly for the uninterrupted power supply and for letting me use their upcoming Architecture Lab facilities. I thank the CASL people, Mrs. Ashalatha, Mr. Shankar and the department for the computing facilities provided. I should also mention my thanks to all the final year ME integrated students and others who were part of the local league cricket, for the fun we had. My stay in IISc has been most memorable due to all these people mentioned here and many more unnamed.

I finally thank my mom, my sister, Periamma, Periappa and Meena Akka' for their patience, support, encouragement and for letting me free to pursue whatever I desired to.

Contents

Acknowledgements **i**

Abstract **viii**

1 Introduction **1**

2 Background **3**

2.1 Unix Process Scheduling Algorithms 4

2.1.1 Bach Scheduler 4

2.1.2 4.3BSD 6

2.1.3 System V Release 4 8

2.1.4 Linux 12

2.2 Real-Time Systems 13

2.2.1 Classification of Real-Time Systems 13

2.2.2 Scheduling Algorithms 14

3 Fairness in Scheduling **16**

3.1 Introduction 16

3.2 Workload 18

3.3 Measure of Fairness 20

3.4 Experimental setup on Linux 21

3.4.1 Results 25

3.5 Experiments with compute bound processes 30

3.5.1 Results . . . 32
3.6 The Phenomenon of Overtaking . . . 42
3.7 Further Comments . . . 49
3.8 Conclusion . . . 50

4 Soft Real-Time Scheduling 51

4.1 Introduction . . . 51
4.2 Background . . . 51
4.2.1 Priority Assignment . . . 52
4.2.2 Scheduling Algorithms . . . 54
4.3 Experimental Setup . . . 56
4.3.1 Linux Signal Handling Bug . . . 56
4.3.2 System Calls Added . . . 57
4.3.3 Workload . . . 58
4.3.4 Fine Tuning . . . 59
4.3.5 Performance Measures . . . 60
4.4 Results . . . 62
4.5 Conclusion . . . 75

5 Conclusion 76

5.1 Results and Contribution . . . 76
5.2 Further Work . . . 77

Bibliography 78

List of Tables

2.1 Different scheduler classes 8

2.2 Dispatch parameter table for the time-shared class 10

2.3 Dispatch parameter table for the real-time class 11

3.1 The Perfect Club benchmarks used. 19

4.1 Baseline parameter values 58

4.2 The weightages assigned for different slack values 61

4.3 Miss ratios for a POSIX system under experimentation and simulation. . . 74

List of Figures

3.1 Bach Scheduler with CPU quanta of 30 millisecs. 23
3.2 Bach Scheduler with CPU quanta of 50 millisecs. 23
3.3 Bach Scheduler with CPU quanta of 100 millisecs. 24
3.4 Bach Scheduler with CPU quanta of 150 millisecs. 24
3.5 Bach Scheduler with CPU quanta of 250 millisecs. 25
3.6 Decaying by 1.5, with CPU quanta of 100 millisecs. 26
3.7 Decaying by 4, with CPU quanta of 100 millisecs. 27
3.8 4.3BSD Scheduler with CPU quanta of 30 millisecs. 28
3.9 4.3BSD Scheduler with CPU quanta of 100 millisecs. 28
3.10 4.3BSD Scheduler with CPU quanta of 250 millisecs. 29
3.11 Priority re-computaion with CPU quanta of 100 millisecs. 30
3.12 Expected CPU distribution in case of a Fair Scheduler. 33
3.13 CPU time distribution: SUN Sparc20 Scheduler. 34
3.14 Magnified view of CPU time distribution: SUN Sparc20 Scheduler. 35
3.15 Fairness measure for SUN Sparc20 's System V Release 4 process scheduler. 36
3.16 Fairness measure for odd number of processes: Bach scheduler. 38
3.17 Fairness measure for even number of processes: Bach scheduler. 39
3.18 CPU usage for six processes and it's fairness measure: Bach scheduler . . . 40
3.19 Scheduler Fairness: seven identical compute bound processes. 41
3.20 Fairness measure for 4.3BSD process scheduler: seven identical compute bound processes. 43
3.21 Overtaking: Bach scheduler. 44

3.22 An erring process among the ten processes that started simultaneously. . . 46
3.23 Overtaking: SUN Sparc20 process scheduler. 47
3.24 No overtaking: 4.3BSD process scheduler. 48
3.25 No overtaking: Bach scheduler with priority re-computation. 48
3.26 Simulated Bach scheduler for the case of ten processes. 49

4.1 POSIX system with 128 priority levels: real-time workload. 63
4.2 Traditional Unix system with 40 priority levels: real-time workload. 65
4.3 Round Robin scheduling in a POSIX system: real-time workload 66
4.4 Tuning the algorithms in a POSIX system: load of 0.3 68
4.5 Tuning the algorithms in a POSIX system: load of 0.65 69
4.6 Re-shift ratios for EDABS on a POSIX system 71
4.7 Simulation of earliest deadline algorithms on a POSIX system 72
4.8 Simulation of least slack algorithms on a POSIX system 73

Abstract

In most modern multi-tasking operating systems the part played by the process scheduler is critical in ensuring that users, at a macro level, as well as processes, at the system level, are treated fairly. The basic functionality of traditional operating systems has already started including real-time capabilities; the effective emulation of real-time scheduling algorithms has therefore become an important issue. There are several advantages to providing real-time support in a general purpose operating system like Unix – for example, ease of code development, availability of a large set of development tools and portability. This work evaluates the performance of decay usage process schedulers through experimentation in the contexts of fairness and support for real-time applications.

We quantify the fairness of decay usage process schedulers and suggest a method to improve the fairness of the normal Unix process scheduler based on detailed experimental studies. We observe that standard assumptions about the start of the decay period and the execution time of processes used in analytical studies of decay usage scheduler are unrealistic. In fact, in our experiments we find that unfairness in the form of overtaking among processes, which could not happen if the assumptions were valid, occurs frequently.

In the second part of this work, an experimental evaluation of emulating soft real-time scheduling algorithms is done by modifying the Linux operating system's source code. The related literature contains simulation studies, but we find that it is extremely difficult to reproduce the workload conditions in an experimental setup. As a result, the simulation and experimental results do need not always agree making simulation an

unsuitable methodology to study this issue. Depending on the evaluation criteria, different algorithms appear to be best; this is neatly captured by two new performance measures that we introduce. Our experimental results show that the default Unix decay usage process scheduling performs comparably to the POSIX process scheduler as long as the mean execution time of tasks is smaller than the CPU quantum assigned per process.

Chapter 1

Introduction

In an operating system, the part played by process management, particularly the process scheduler, is very important; it not only manages the various processes of the users and the interaction between them, but also enables these contending processes to share the CPU *fairly* amongst themselves, while ensuring that system throughput and response times are good.

In this context, various scheduling algorithms have been proposed. The round robin algorithm assigns each runnable process a fixed CPU quantum, and all the processes are serviced in a round robin order. A process is assigned the CPU, and when the CPU quantum elapses, it is switched out and the next process in the queue is assigned the CPU. An improvement on this algorithm is the multi-level round robin, where there is more than one level of ready queues in which the process can be. The higher the level, the smaller the CPU quantum assigned, and the greater the priority for getting the CPU. Whenever a process exceeds it's CPU quantum, it is switched out of the CPU and also moved to a lower level of the scheduling hierarchy. This algorithm has a drawback, in that a process at a lower level might never get the CPU due to the continuous presence of processes in higher levels, which is clearly unfair.

In general time sharing systems, including Unix, use some kind of decay usage scheduling algorithm which assigns priority to a process based on it's CPU usage. This CPU usage value is decayed (reduced in value) periodically in order to reflect the recent pattern

of CPU usage of a process. The priority value assigned to a process is inversely proportional to its CPU usage, and the highest priority process is always assigned the CPU. In this way, the scheduler can avoid the above mentioned drawback and be fair to all the processes present in the system. The round robin with multilevel feedback scheduling algorithm, which we discuss in more detail in the next chapter, comes under this class of decay usage schedulers. All decay usage schedulers loosely address the issue of fair treatment of different kinds of processes - for example, interactive as well as batch processes. However, the existing literature does not address how fair they actually are. This is precisely what we have addressed in the first part of this work, by defining a *fairness measure* and running experiments on a system with real workloads.

The second part of this work is an experimental evaluation of scheduling algorithms that have been developed for soft real-time systems. Real-time systems are typically stand-alone systems, built to specifically control and support the real-world environment in which they operate. Tasks in such systems are time critical and their scheduling plays an important role in achieving the timing constraints of various tasks. Earliest Deadline (ED) and Least Slack (LS) are the two general classes of scheduling algorithms widely used in real-time systems. The literature contains some work on emulating these algorithms in soft real-time systems, and evaluating them through simulations, but none on their experimental evaluation.

This thesis is organised as follows: Chapter 2 describes the decay usage schedulers used in various Unix systems, as well as scheduling algorithms used in real-time systems. Chapter 3 surveys the literature related to the fairness aspects of process schedulers, and describes our experimental work on the fairness of decay usage schedulers. Chapter 4 describes our experimental work on the use of decay usage schedulers in soft real-time systems. We conclude in Chapter 5, summarising the results obtained and suggesting related areas for possible further exploration.

Chapter 2

Background

The process scheduler is the part of an operating system that manages the sharing of CPU time among potentially many competing processes. In a time sharing system the process scheduler allocates the CPU to a process for a period of time called a time-slice or time *quantum*. It later preempts the process and schedules another when that time slice expires. In the Unix operating system, every active process has a scheduling *priority* associated with it and the kernel always does such context switches to the highest priority process at that moment.

The Unix process scheduler has many objectives to satisfy, some of them contradicting each other. These include high system throughput, efficient utilisation of resources, fairness and quick response to processes. Unix uses the category of schedulers based on *round robin with multilevel feedback* - whenever the kernel preempts a process, it feeds it back into one of n scheduling priority levels. The priority values are recalculated periodically, depending on the recent activity of the process, providing *feedback*. For all processes in the same priority level, the CPU is assigned in a *round robin* manner. As the scheduling decisions are taken with respect to the current priority of a process, it is only fair that the priority is inversely dependent on the *recent* CPU usage of the process. Thus, the priority of a waiting process increases (becomes numerically lower), while that of a running process decreases. In this chapter, we review background literature in two areas - Unix decay usage schedulers and scheduling considerations in soft real-time systems.

2.1 Unix Process Scheduling Algorithms

We next look into the basic Unix decay usage process scheduling algorithm and then discuss the variations used in 4.3BSD, System V Release 4 and Linux process scheduling algorithms. The basic Unix process scheduling algorithm is described in detail by Bach [3]. We will refer to it as the Bach Scheduler throughout this report.

2.1.1 Bach Scheduler

In Unix, process priorities fall within two ranges: the *user-level* priorities are those numerically above a specific threshold value, and the *kernel-level* priorities are below the threshold value.[1] Processes are normally executed with user-level priorities, and the kernel-level priorities are attained in the sleep algorithm. The kernel-level priorities are further divided into two groups: processes with low kernel priority wakeup upon the receipt of a signal, while processes with high kernel priority continue to sleep uninterrupted.

A fixed kernel-level priority value is assigned to a process before it goes to sleep, depending on the reason for sleeping. This value is hard-coded and does not depend on the run-time characteristics of the process. A process that sleeps in low-level system code tends to cause more bottlenecks; it is therefore given a better priority (a numerically lower priority value) than a process that would cause fewer system bottlenecks. Thus, a process waiting for a disk I/O has a better priority than a process waiting for a free buffer, as it already has a buffer and there is a good chance that it will free the buffer as soon as it finishes the I/O. The kernel adjusts the priority value of a process that returns from kernel to user mode and also penalises it, as it has just used valuable kernel resources.

While a process is using the CPU, it's *recent_cpu_usage* value is incremented once every clock tick. Further, once every decay period (typically, once every second) the

[1]Unless indicated otherwise, the higher the value of priority for a process, the lower it's perceived importance. In the discussion that follows, we sometimes refer to a process getting *better* priority - meaning a lower value, that is more conducive to it's receiving the CPU soon. Where necessary we use an unambiguous notation, clearly specifying whether priority values are high or low relative to each other.

recent_cpu_usage of each process is decayed by a decay factor (typically, 2), i.e.,

$$recent_cpu_usage \;=\; recent_cpu_usage/2$$

The priority of a process is recomputed as:

$$priority \;=\; (recent_cpu_usage/2) + base_level_user_priority$$

where the *base_level_user_priority* is the threshold value between user and kernel level priorities described above. The order and manner in which the scheduling of different processes is carried out play an important part in achieving the objectives of the Unix process scheduler mentioned above. This is entirely dependent on the priority value of a process, which in turn is dependent on the activity of the process and the way the activity was monitored through the scheduling parameters, viz, recent cpu usage, decay factor, decay period and the priority function which maps the recent cpu usage to a priority value.

Let T_i be the amount of CPU time received by a process in clock ticks in the i th decay period. Then, in consecutive decay periods, the *recent_cpu_usage* is updated as follows,

$$\begin{aligned}
recent_cpu_usage &= 0.5 \times T_0 \\
recent_cpu_usage &= 0.5 \times (T_1 + 0.5 \times T_0) \\
&= 0.5 \times T_1 + 0.25 \times T_0 \\
recent_cpu_usage &= 0.5 \times T_2 + \ldots + 0.125 \times T_0
\end{aligned}$$

with priority being re-computed as,

$$priority \;=\; (recent_cpu_usage \times 0.5) + base_level_user_priority$$

Observe that after three decay periods from now, in calculating the priority value, the weightage associated with the cpu time consumed (T_0) in the current decay period is

about 12 percent. The algorithm thus forgets almost 90 percent of the process' current activity within the next three decay periods. We refer to this period (in this example, three decay-periods) in which 90 percent of the current process behaviour is forgotten as the *spread* of the scheduling algorithm. Thus, the value of *spread* for the Bach scheduler is 3. We will later use the value of *spread* in discussing the fairness properties of various schedulers.

2.1.2 4.3BSD

In 4.3BSD Unix [14], process scheduling is as described above, except that the decay factor is a function of the load present on the system at that time. The process's priority value is based on the variables *p_cpu* and *p_nice* maintained in the *proc* kernel data structure associated with every process. The value of *p_nice* is in the range -20 to $+20$, with negative values providing higher priority to the process. The default value of *p_nice* is zero. *p_cpu* takes care of the recent CPU usage of the process; it is incremented once every clock tick if the process is found to be executing at that time. It is also decayed once every second as:

$$p_cpu = [\frac{2 \times load}{2 \times load + 1}] \times p_cpu + p_nice$$

where, *load* is the sampled average number of processes waiting for the CPU in the past 1 minute interval. Every time the *p_cpu* field is incremented, the value is checked to see whether it is a multiple of four. If so, the process priority value is re-calculated as:

$$priority = PUSER + p_cpu/4 + 2 \times p_nice$$

Every process is allocated a quantum of 100 milliseconds and is context switched out at the end of a quantum, in favour of the highest priority (numerically lowest) process in the run queue. Once every second, the priority of each runnable process is recalculated according to the above decay equation. For efficiency reasons the following optimisation is done with respect to blocked processes waiting for an event to happen: Since blocked

processes cannot accumulate *p-cpu*, there is no point in decaying their values once every second. Instead a variable *p_slptime* is incremented once every second for all the blocked processes, and when the process is awakened the value of *p_cpu* is adjusted as:

$$p_cpu = [\frac{2 \times load}{2 \times load + 1}]^{p_slptime} \times p_cpu$$

Assume that there is only one process in the system. The assumption is necessary as the load average is part of the decay function in this algorithm. Consider the *spread* of this process scheduling algorithm with *p_nice* at the default value of zero. Here, the value of *p_cpu* is successively updated as follows:

$$\begin{aligned}
p_cpu &= (2 \times 1) \times T_0/(2 \times 1 + 1) \\
&= 0.66 \times T_0 \\
p_cpu &= 0.66(T_1 + 0.66 \times T_0) \\
&= 0.66 \times T_1 + 0.44 \times T_0 \\
p_cpu &= 0.66 \times T_2 + 0.44 \times T_1 + 0.30 \times T_0 \\
p_cpu &= 0.66 \times T_3 + \ldots + 0.20 \times T_0 \\
p_cpu &= 0.66 \times T_4 + \ldots + 0.13 \times T_0
\end{aligned}$$

After a period of 5 seconds, we see that only 13 percent weightage is assigned to the CPU time consumed in the current quantum, i.e. almost 90 percent of the CPU time consumed up to now is forgotten, resulting in a *spread* of about 5. Note that for this algorithm, the notion of *spread* is not very well defined, as the decay function depends on the number of processes in the system. For example, if there are five processes in the system, the *spread* will be about 17 and it increases further as the number of processes increases. This is due to the fact that the decay rate is slower under higher system loads. Taking into account the way the priority value of a process is calculated from the *recent_cpu_usage* value in 4.3BSD, we see that the *spread* is 3 for the single process case and 13 for the five process case.

Since the decay rate is slow, processes do not jump across priority levels upon decaying, but step through the levels one by one. As a result, as the *spread* value increases, a given set of processes tends to occupy more levels than with the previously described basic Unix scheduling algorithm. Thus, what the larger *spread* amounts to is, a finer difference among the processes being maintained by assigning different priority levels to them, rather than clubbing them into the same level and treating them equally.

2.1.3 System V Release 4

Until now, we have seen process schedulers where all processes were subject to the same selection criteria in assigning the CPU. The notion of priority classes is introduced in Unix System V Release 4 [9]. There are three priority classes: *Real time*, *System* and *Time shared*. Each class has a set of *class_dependent* routines associated with it, which calculate the priority level of a process and place it in the appropriate priority queue. The kernel's process selection code - part of the class independent routines - selects a process from the highest valued priority queue and assigns the CPU to it.

Priority Class	Scheduling Sequence	Global Priority Value
Real time	First	159
	.	.
	.	.
	.	.
	.	100
System	.	99
	.	.
	.	.
	.	.
	.	60
Time shared	.	59
	.	.
	.	.
	.	.
	Last	0

Table 2.1: Different scheduler classes

Table 2.1 shows the order of importance of the different classes and the range of priority values for each class. Note that the higher the global priority value, the better the priority of a particular process. Thus, real-time processes have higher (numerically higher) priority than system class processes, and system class processes have higher priority than time-shared processes. By default, all processes initiated by users are assigned to the time-shared class. The system class consists of system processes, such as the scheduler, swapper, etc., which run with a fixed priority. This class is not configurable and is reserved for kernel use only. A user process running in *kernel mode* is not the same as a system class process, as it runs with it's own scheduling characteristics.

Priority class groups

All processes are arranged into specific priority class groups and each group is categorised by it's own scheduling characteristics determined by the class dependent functions. These functions determine a process's priority value, which is made available to the class independent kernel functions as a global priority value. The highest valued global priority is chosen by the scheduler which runs the process from the head of the highest global priority dispatch queue. The process runs until it uses up it's time slice, is pre-empted by a higher priority process, or blocks waiting on an event.

When a process is created, it inherits it's parent's scheduling parameters, which include the priority class and the priority value within that class. It remains in the same class unless changed as a result of a user request through *priocntl* command or system call. Each priority class maintains it's own table of values to describe the characteristics of a process, known as *dispatch parameter table*. Table 2.2 is one such table provided as default for time-shared class of processes, where

globpri is the priority value assigned to a process. The initial value is 59 for a user process.

quantum is the value of time quantum (in millisecs) for which the CPU is assigned to a process.

tqexp is the new process priority, if the process uses up all of the time quantum assigned

globpri	quantum	tqexp	slpret	mwait	lwait	globpri	quantum	tqexp	slpret	mwait	lwait
0	200	0	50	0	50	30	80	20	53	0	53
1	200	0	50	0	50	31	80	21	53	0	53
2	200	0	50	0	50	32	80	22	53	0	53
3	200	0	50	0	50	33	80	23	53	0	53
4	200	0	50	0	50	34	80	24	53	0	53
5	200	0	50	0	50	35	80	25	54	0	54
6	200	0	50	0	50	36	80	26	54	0	54
7	200	0	50	0	50	37	80	27	54	0	54
8	200	0	50	0	50	38	80	28	54	0	54
9	200	0	50	0	50	39	80	29	54	0	54
10	160	0	51	0	51	40	40	30	55	0	55
11	160	1	51	0	51	41	40	31	55	0	55
12	160	2	51	0	51	42	40	32	55	0	55
13	160	3	51	0	51	43	40	33	55	0	55
14	160	4	51	0	51	44	40	34	55	0	55
15	160	5	51	0	51	45	40	35	56	0	56
16	160	6	51	0	51	46	40	36	57	0	57
17	160	7	51	0	51	47	40	37	58	0	58
18	160	8	51	0	51	48	40	38	58	0	58
19	160	9	51	0	51	49	40	39	58	0	59
20	120	10	52	0	52	50	40	40	58	0	59
21	120	11	52	0	52	51	40	41	58	0	59
22	120	12	52	0	52	52	40	42	58	0	59
23	120	13	52	0	52	53	40	43	58	0	59
24	120	14	52	0	52	54	40	44	58	0	59
25	120	15	52	0	52	55	40	45	58	0	59
26	120	16	52	0	52	56	40	46	58	0	59
27	120	17	52	0	52	57	40	47	58	0	59
28	120	18	52	0	52	58	40	48	58	0	59
29	120	19	52	0	52	59	20	49	59	32000	59

Table 2.2: Dispatch parameter table for the time-shared class

to it.

slpret is the new process priority assigned, upon a return from sleep.

mwait is the number of seconds within which the process must use it's assigned time slice. If not, it's priority is set to *lwait*. This ensures that there is no starvation of low priority processes.

lwait is the new priority value for a process that has exceeded the maximum wait time of *mwait* seconds.

The time slice assigned is of variable size, and is dependent on the priority level of the process. Larger time slices are given to processes with lower priorities and vice versa. Thus, although a low priority process is likely to starve for some time, once it gets the CPU it receives a large chunk of CPU time. The *mwait* and *lwait* parameters combine to increase the priority of a process that has not received the CPU for some pre-determined time. The *tqexp* parameter is used to ensure that a process which uses the CPU continuously gets lower (numerically lower) priority. Entering sleep states frequently is a characteristic of interactive processes; hence the high priority value for a process that returned from the sleep state.

rt_globpri	rt_quantum	rt_globpri	rt_quantum	rt_globpri	rt_quantum	rt_globpri	rt_quantum
100	1000	115	800	130	400	145	200
101	1000	116	800	131	400	146	200
102	1000	117	800	132	400	147	200
103	1000	118	800	133	400	148	200
104	1000	119	800	134	400	149	200
105	1000	120	600	135	400	150	100
106	1000	121	600	136	400	151	100
107	1000	122	600	137	400	152	100
108	1000	123	600	138	400	153	100
109	1000	124	600	139	400	154	100
110	800	125	600	140	200	155	100
111	800	126	600	141	200	156	100
112	800	127	600	142	200	157	100
113	800	128	600	143	200	158	100
114	800	129	600	144	200	159	100

Table 2.3: Dispatch parameter table for the real-time class

The real-time class of processes is assigned a fixed priority value and when the time quantum for a process expires, it is reassigned that same priority level. As long as there is a runnable real-time class process in the dispatch queue, no other system class or time-shared class process is assigned the CPU. The dispatch parameter table for real-time class is shown in Table 2.3 where,

rt_globpri is the priority value assigned to a real-time process.

rt_quantum is the value of time quantum in millisecs, for which the CPU is assigned to a process.

As real-time process priority values are higher than even system class processes, a careful use of real-time privileges is required.

2.1.4 Linux

Linux (as of Release 1.2.3) has, by far, the simplest scheduling algorithm among those surveyed here. All scheduling decisions are based on the *proc* structure variable called *counter* associated with every process. *counter* is the number of *jiffies* [2] allotted to a process once it gets the CPU. The *counter* value decreases with every system clock tick, as long as the process is running, until it reaches zero. Then, the schedule function is called to select the runnable process with the highest non-zero counter value. If the counter value for no existing processes is greater than zero, new counter values are assigned for *all* the processes, as

$$counter = counter/2 + priority.$$

For the *init* process, the initial priority value is 15 and the initial value of counter is 15. Whenever a process is forked off, the initial priority is the same as that of it's parent and the counter value is assigned as

$$counter = counter/2 + priority$$

[2] 1 jiffy = 10 millisecs

Thus, the *counter* variable acts as a time slice and also as a priority value for a process; the higher the value, the higher the priority of the process.

2.2 Real-Time Systems

A *real-time* system is one which provides support for tasks that are meant to be executed within a given timing constraint. By definition [8], 'A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred.' A good example is a robot that has to pick up a piece from a conveyor belt. The piece is moving, and the robot has a small window of opportunity in which to pick up the object. If the robot is late, the piece will not be accessible, and thus the job will have been done incorrectly even though the robot went to the right place. If the robot is early, the piece will not be there yet, and the robot may block it.

Unlike normal systems, real-time tasks have pre-determinable execution times and resource requirements. The real-time task has to be completed within the given deadline if its execution is to have any meaning at all. Also, a *value* function is associated with every real-time task. The value is normally positive, with higher values meaning that the task is of higher importance. As soon as the deadline for the completion of a task is over, the value can become negative, zero or decrease towards zero as time elapses. A negative value signifies that the task just missed was a critical task and the result of this miss might be disastrous. A zero value means that the task is of no significance to the system any longer and can be abandoned.

2.2.1 Classification of Real-Time Systems

Real-time systems are classified into three categories based on the characteristics of the *value* function, *viz*, *hard*, *firm* and *soft* real-time systems. The tasks in *hard real-time* systems have deadlines which are hard, meaning that if the real-time task fails to complete

its execution before the deadline, the results would be disastrous. Thus, the usefulness *value* of the task takes on the maximum negative value after the deadline is over.

In *firm real-time systems* if the real-time task fails to completes its execution within the deadline, the *value* becomes zero and the task is of no use. Hence, the scheduling algorithm does not allow the completion of this task, and it is abandoned. Such systems also guarantee that the deadlines for critical tasks are always met, regardless of the load present on the system. Thus, scheduling algorithms on such systems typically stop the execution of a task whose deadline is past, and switch to other tasks which can be completed within their deadlines. The scheduling algorithm chooses to start the execution of a task only if it is possible to complete it within the given deadline. Also, in most *hard* and *firm* real-time systems, scheduling is optimised for periodic tasks, and requires that the maximum execution time of a task is known *a priori*.

On the other hand, in a *soft real-time* system, the scheduler requires only an estimate of the execution times, or in most cases no such knowledge is required at all. In these systems, it is difficult to guarantee that deadlines will be met, and hence the scheduling algorithm tries to minimise the number of missed deadlines. Difficulties arise from the nature of the system. For example, in a database system, the completion of a request is dependent on the number of transactions that are concurrently running and holding locks. Also, disk access times will be dependent on where the previous request left the disk arm, which too is unpredictable.

2.2.2 Scheduling Algorithms

For any given real-time task, a deadline within which it has to complete must be provided along with an estimate of the execution time upon it's arrival. The difference in time between the deadline and the estimated execution time is termed as the *slack* for that task. This slack value in a way provides an indication of the importance of task in hand; the higher the slack value, the lesser is the process' importance compared to other tasks. With respect to the deadline given, it can be inferred that the earlier the deadline, the greater the importance of the task. Based on these facts, two classes of scheduling algorithms are

used in real-time systems:

Earliest Deadline (ED) - Here, the highest priority is assigned to the task with the earliest deadline, which is allowed to use the CPU. ED performs optimally in systems which are not overloaded. In real-time systems, as and when a task arrives, it's deadline is compared with the currently executing task's deadline. If the deadline is earlier than the current task's deadline, then the current task is preempted and the newly arrived task is assigned to the CPU. The remaining tasks are sorted in ascending order by their deadline and put into a queue. Whenever the CPU becomes free the first task in the queue is assigned the CPU.

Least Slack (LS) - Here, the highest priority is assigned to the task with the lowest slack value. The slack value is determined at the time of task arrival and is not changed thereafter. Upon arrival of a new task or upon completion of a task, the task with the least slack value is assigned the CPU.

Chapter 3

Fairness in Scheduling

3.1 Introduction

One of the stated goals of Unix decay usage process schedulers is *fairness*. The scheduling literature contains some work that pays special attention to the issue of fairness. The Fair Share Scheduler described by Henry [11] is one such effort. It attempts to ensure that CPU time is divided among the set of users according to pre-determined ratios. It clubs users into groups known as *fair share groups*, with each group guaranteed a share of CPU time, irrespective of the load on the system. This is intended to prevent a sudden burst of activity by one class of users, say students with a deadline on a course assignment, from occupying the whole system and rendering it useless to other users. Henry's implementation associates a share penalty with the usual priority calculation done by the scheduler, viz,

$$FSS_priority \quad = \quad Unix_priority + share_penalty$$

where,

Unix_priority is a value proportional to the recent CPU usage of the process,

share_penalty is a value proportional to the recent fair share group CPU usage and

FSS_priority is the new priority value of a process, based on which scheduling decisions are taken.

Thus, the priority of a process worsens (i.e., its value increases) as either that process

or other processes in the same share-group get more CPU time.

The SHARE scheduler [13] is another extension to the Bach scheduler with fairness in mind. Its goal is to ensure long term fairness to share-groups. A fairness interval, often measured in days, is used, with sampling done every 4 seconds. Like the Fair Share Scheduler, SHARE adds a share penalty in priority calculation, but the penalty is scaled by the number of active processes in that share-group, while ensuring that the short term usage does not exceed the long term allocation by too much. SHARE also supports hierarchical fair share groups. The PrismaOS scheduler described by Essick [6] works along similar lines, except that it is an event based scheduler. One of the objectives in PrismaOS scheduling is to handle thousands of processes with as low an overhead as possible. It ensures accurate short term fairness with a fairness interval of 15 seconds and sampling done every second.

Hellerstein developed an analytical model for decay usage scheduling which relates the service rates of compute-bound processes to their base priorities and scheduler parameters [10]. The fundamental idea in his work is that periodically changing the priority of a process with respect to its CPU consumption and the service rate objectives for that user, will increase scheduling overheads considerably if the objective is to be achieved with precision. Instead, this overhead is avoided by exploiting an already existing feed-back loop in the scheduler, i.e., priority being a function of recent CPU usage of the process, to calculate only the required base priorities of processes in order to achieve the desired service rates for all classes of processes. This idea can also be used in fair share schedulers to obtain the most reasonable base priority for a particular share-group. However, Hellerstein's model considers only compute bound processes, not considering interactive or I/O related activity, and is not directly applicable to real world conditions. Epema extended Hellerstein's analysis technique to decay usage scheduling in multiprocessor systems [5], showing that (i) the relationship between the shares achieved and the number of jobs in the classes is subtle, and (ii) among the schedulers studied, 4.3BSD provides the highest level of control over the share ratios.

All of the work reviewed above views fair process scheduling from the user's viewpoint.

In this chapter, we take the alternative approach of viewing fairness from the process' perspective. Our studies of fairness were conducted through actual measurements on real systems. We first describe the benchmark workload that we used in our experimental work. It was designed to be a realistic representative of the load on workstations at our computer centre. We then look at alternative methods to quantify fairness, providing justifications for the fairness measures that we finally use. Finally, we describe our experimental setup and results.

3.2 Workload

To understand the fairness characteristics of decay usage process schedulers, we must consider realistic workloads containing a mix of the various kinds of processes found on a real system, unlike the simplistic workload assumptions made by Hellerstein [10] and Epema [5]. We base our workload selection on data we gathered through system activity monitoring experiments, extending on the studies of Shet [16]. Shet monitored the load on a system continuously, classified the workload in terms of a few parameters, and predicted the response time of a process by extending the recent load conditions of the system to the future. His process classification differentiated between

1. interactive processes, which have CPU usage on the order of few milliseconds,
2. long-lived processes, which have bursty CPU requirements, and,
3. back-ground processes, which have continuous CPU requirement.

To obtain workloads for our fairness experiments, we first monitored the workload on a MIPS R2000 Personal IRIS system running IRIX 4.0.5, in a networked workstation environment at the Supercomputer Education and Research Centre at IISc. The monitoring program periodically obtained statistics on all the processes present in the system. In order to do so, the following kernel data structures were monitored: the *process table*, the *uarea* and the *sysinfo*. The process table contains an entry for each process with information, accessible to the kernel at all times, such as current process status, pid, priority,

recent CPU usage, etc. The *uarea* (user area) contains fields that are accessible by the kernel while the process is running. It includes the total CPU usage of the process in user mode and system mode, the total number of reads and writes done by it, the number of voluntary and involuntary context switches so far, etc. The *sysinfo* structure contains a system wide collection of statistics maintained by the kernel, including the number of forks executed, the time spent by the system in user mode and system mode so far, etc. As the *uarea*, only contains information about the running process, it does not reside in un-swappable kernel main memory. In other words, while the entire process table forms part of the kernel address space, the *uarea* of only the running process is mapped to a known location in the kernel address space. Every process table entry contains a pointer to the respective *uarea*, which resides in main memory and can be swapped out.

Based on Shet's [16] observations and the confirmation of the same from our monitoring work, we arrived at a workload of: two compute bound processes (doing no I/O), five compute intensive processes (with varying degrees of I/O and memory occupancy), one script to capture the effects of interactive users, one memory bound and one I/O bound process. The script for interactive users periodically does some editing using *vi*, compilation of small C programs, recursively does a *find*, does *ps*, *ls*, etc. The five compute intensive processes comprise the four Perfect Club benchmarks [4], CSS, APS, TIS, NAS,

Name	*Description*	*Executable Size*	*Input, Output*
CSS	Circuit Simulation, Spice.	565 KB	16, 32 KB
APS	Fluid Dynamics, using FFT	837 KB	11, 205 KB
TIS	Integral transforms for application in the areas of Chemical and Physical models.	30 KB	5, 1 KB
NAS	sequential Fortran version of the NAS kernels.	248 KB	2.8, 1.4 MB

Table 3.1: The Perfect Club benchmarks used.

shown in Table 3.1, and a *Gzip* script which un*tar*s, *gunzip*s, then *gzip*s and *tar*s an expanded file of about 26 MB in size containing the source code of *emacs-19.25*. Note that the NAS program reads in an input file of size 2.8 MB during execution. The workload thus involves some amount of I/O and memory occupancy, apart from heavy usage of the

CPU.

Our memory intensive program occupies about half of the available main memory by repeatedly touching the start and end of each page of a dynamically allocated 8 MB region. This ensures that the page is brought into main memory even if it had been swapped out. The I/O intensive program un*tar*s and *tar*s a *tar*red file of about 50 MB size, containing the source code of *xemacs-19.13*.

3.3 Measure of Fairness

The next issue we address is how to quantify the fairness of a scheduler. We view fairness from the process viewpoint. To say that one scheduler is *fairer* than another, we must define the term *fair* from the process point of view. The simplest definition would be that given a set of similar processes running under similar workload conditions, the treatment they get should be the same. Thus, they should all be equally starved or satiated with the system resources that they require during the course of the run. Also, across repeated runs of the same experiment their treatment should be consistent. It might happen that sometimes the scheduler does well and sometimes it does not; the treatment that a process receives should be more or less consistent throughout a particular run. But, this is a simplistic view, given that we typically have a system with dissimilar processes with varying degrees of resource requirements for the CPU, memory and I/O.

For any program, the elapsed time is a major factor of concern, as it is only when a program completes that it is of any use. For compute intensive, I/O bound and memory bound applications we are interested in their total response time. For programs with bursty CPU requirements and for interactive processes, we are interested in seeing that CPU requests are satisfied as soon as possible. This can be ensured by associating the best priority with processes that go to sleep and to processes with a high ratio of idle time to CPU time.

We ran measurement experiments as follows for each setting of scheduler parameters: the experimental workload is allowed to run to completion and the elapsed times of all processes noted for repeated runs of the same experiment. Later, the individual component

programs of the workload are allowed to run under *no-load* conditions and elapsed times noted. If the scheduler is fair to all processes, we might expect the ratio of no-load elapsed time to normal elapsed time of each program, which we term as the *blow-up ratio*, to be almost the same for all similar natured programs. The two CPU bound programs are considered to be similar in nature, as are the five compute intensive programs from our test suite.

We justify this measure of fairness as follows: if a particular program is affected by the presence of other processes in the system, the same kind of effect should be felt by similar programs which start at the same time. In order to determine whether the treatment they receive is consistent, we calculate the standard deviation of the *blow-up ratio* values for every program, across different runs. To nullify the effects of different elapsed times for different programs, we compute the standard deviation divided by the average value of the *blow-up ratio*. The scheduler for which these ratios are the same for all the concerned processes and for which the normalised standard deviation value is zero will be considered to be the *fairest* scheduler. In short, our fairness measure F for a process p is computed based on measurements from n executions of the concerned program as

$$F_p = \frac{\sigma \; (Blowup_Ratio_p)}{(\sum_{j=1}^{n} Blowup_Ratio_{p,j}) \; / \; n}$$

where, $Blowup_Ratio$ of a process p on it's j^{th} run is

$$Blowup_Ratio_{p,j} = \frac{Elapsed_Time_{p,j}}{No_Load_Time_p}$$

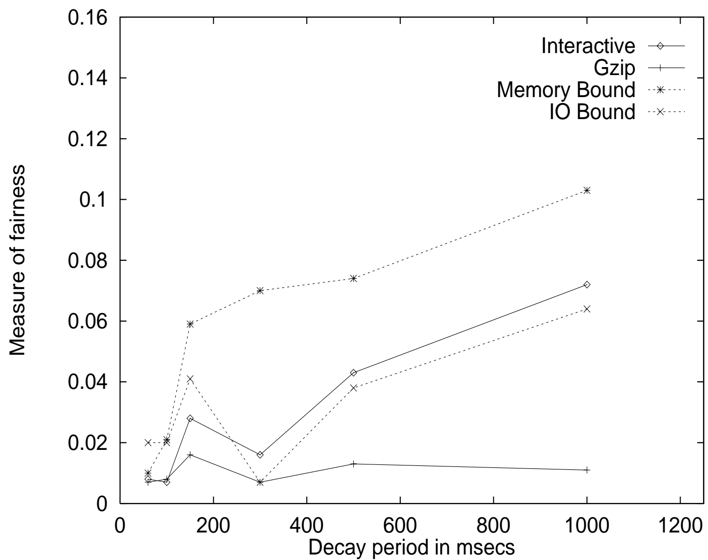
3.4 Experimental setup on Linux

We conducted our fairness studies through measurement, not simulation. This required a system setup which provides the ability to change scheduling parameter values. Most commercial operating systems provide only the ability to change the CPU quantum size, and not the decay rate or the decay period. As we saw earlier, Unix System V Release 4 is

an exception. Even in SVR4, we have only indirect control over the scheduling parameters; there is no obvious correlation between the dispatch table values and the scheduling parameters (decay rate and decay period). Instead, we opted for the freely available public domain operating system Linux. We added control of the scheduling parameters through source code modifications [12]. We have already seen that the scheduling algorithm of Linux is simple, primarily intended for a small number of processes.

Hence, our first task was to implement the Bach scheduling algorithm on top of existing Linux facilities for process scheduling. We determined that the kernel is guaranteed to get control of the CPU in the *do_timer()* routine, which is called HZ times every second. HZ is a configurable system constant whose value is normally 100, giving a system clock tick of 10 milliseconds. On return from any system call, and also whenever *do_timer()* finishes execution, the variable *need_resched* is checked. If it is set, the *schedule* function is called to schedule a different process if so warranted. We modified the *do_timer()* routine to do accounting of recent CPU usage for a process, elapsed CPU quantum, priority calculation and other functionality related to a decay period being over, or end of quantum. Subsequently, we set the *need_resched* flag whenever the current scheduler conditions warrant a switch to another process.

Apart from these changes to the Linux source code, we also implemented a penalty for every process that goes into kernel mode and uses kernel resources. This penalty, as suggested by Bach [3], is implemented by adding the amount of time spent in kernel mode to the priority of a process when it returns to user mode. In addition, to allow scheduling parameters to be changed without the need for re-booting or re-compiling the kernel, we introduced system calls to change the scheduling parameters dynamically. The 4.3BSD type process scheduler was also implemented along with necessary system calls to change the scheduling parameters dynamically. The scheduler has a dynamic decay rate calculation based on the load present in the system, both of which were calculated once every 5 secs.



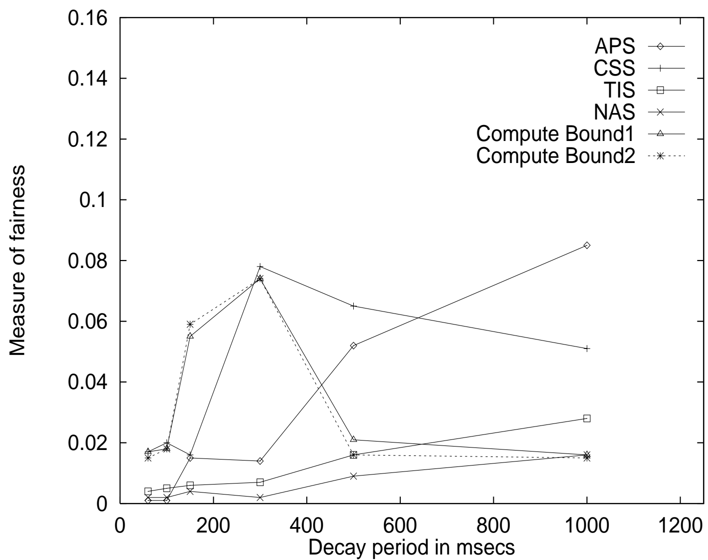
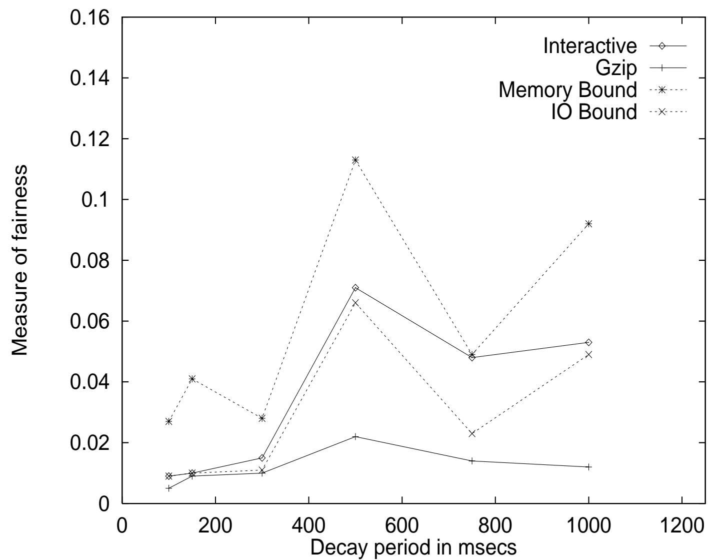


Figure 3.1: Bach Scheduler with CPU quanta of 30 millisecs.



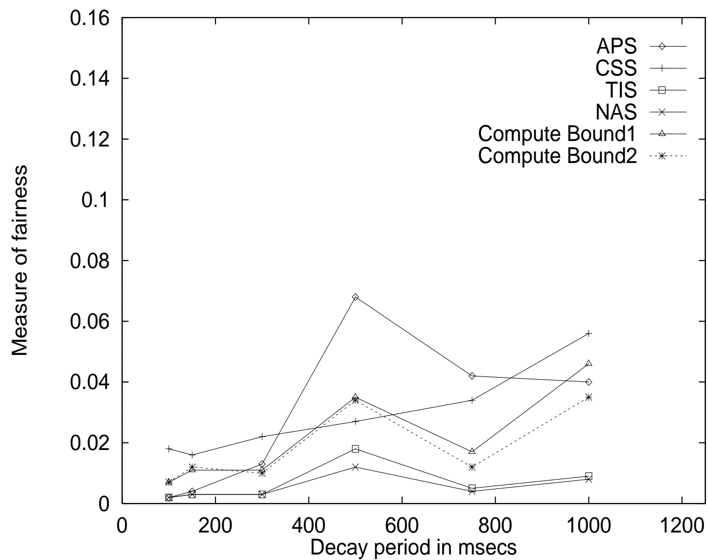
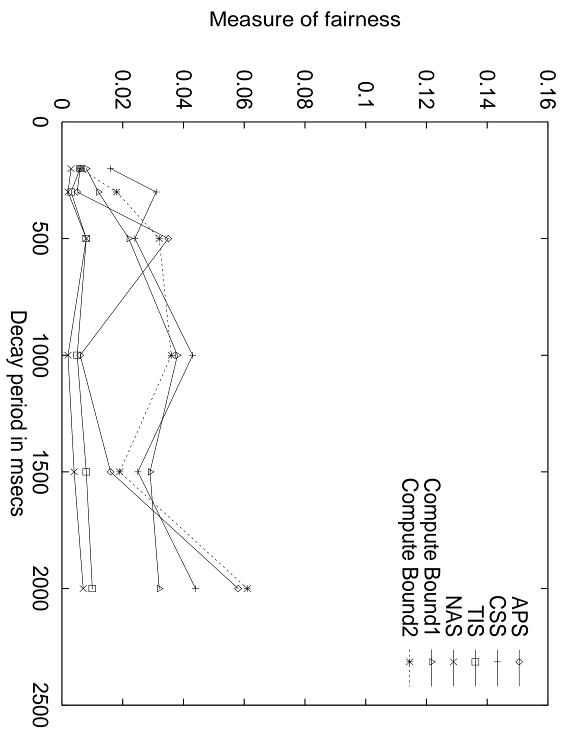


Figure 3.2: Bach Scheduler with CPU quanta of 50 millisecs.



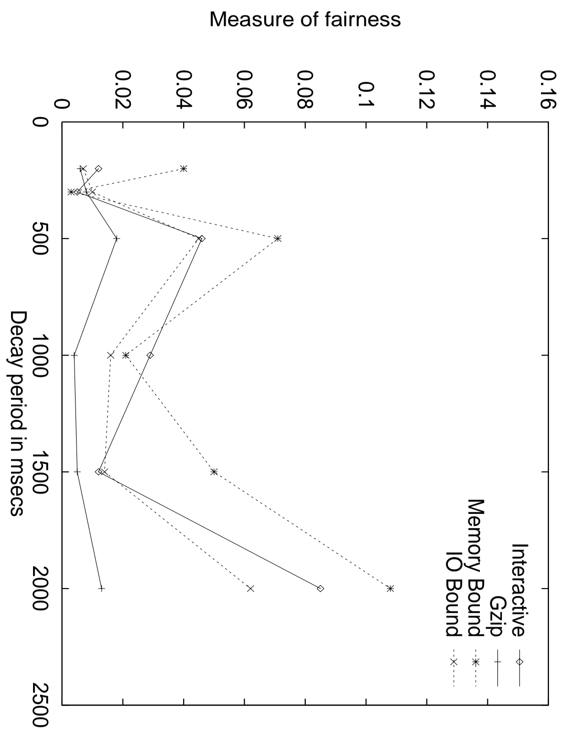
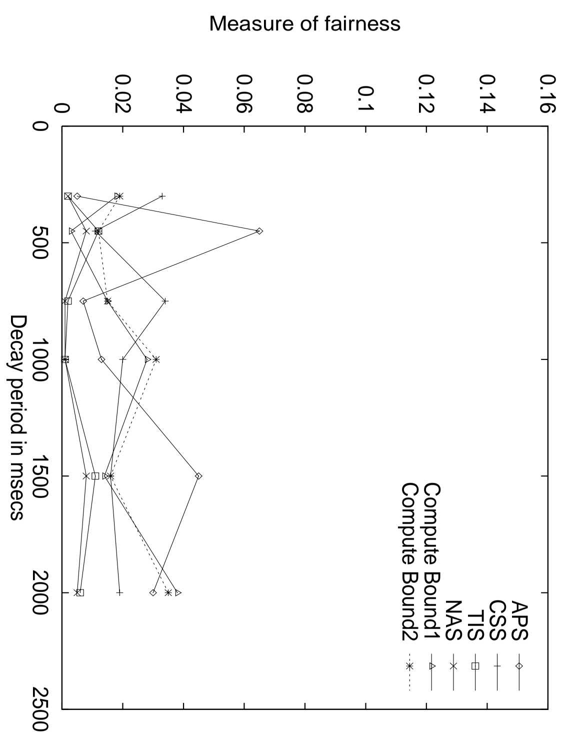


Figure 3.3: Bach Scheduler with CPU quanta of 100 millisecs.



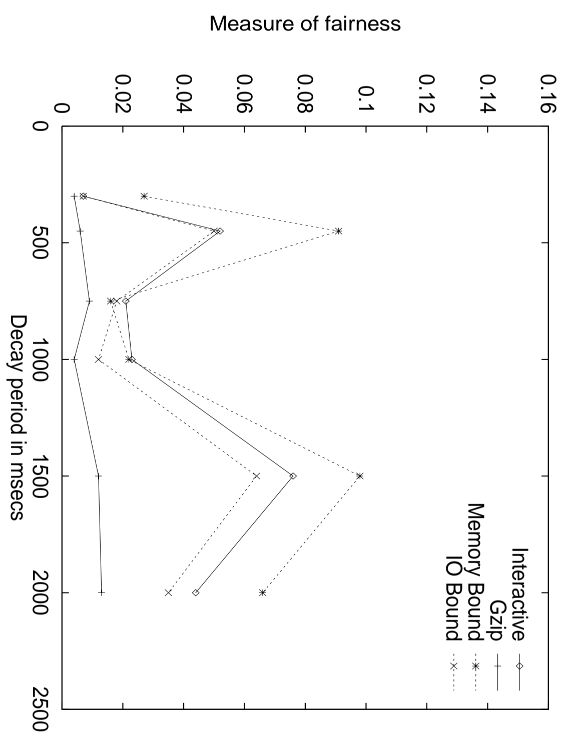


Figure 3.4: Bach Scheduler with CPU quanta of 150 millisecs.

3.4.1 Results

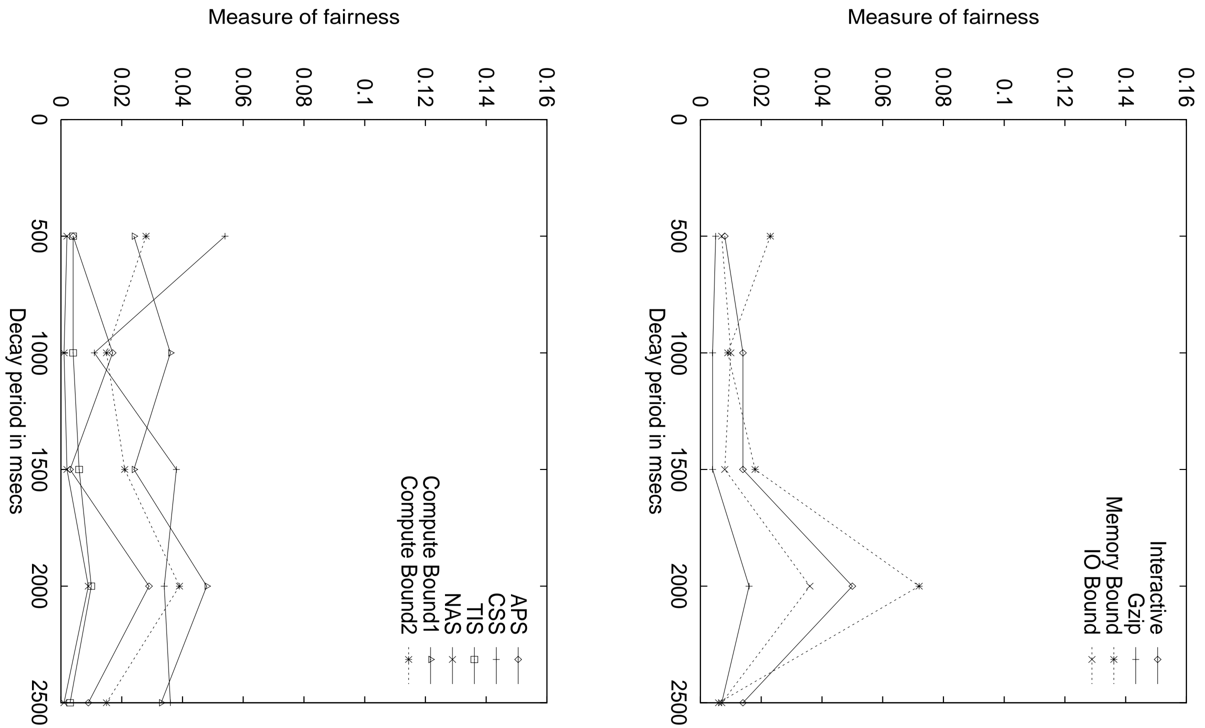


Figure 3.5: Bach Scheduler with CPU quanta of 250 millisecs.

Figures 3.1 to 3.5 show fairness results on varying the CPU quanta assigned per process and the decay period over which the decaying of the *recent_cpu_usage* is done. In these graphs, the decay rate value was 2; once every decay period the *recent_cpu_usage* value is divided by 2. Each point in the graph is the average obtained by repeating the experiment four times. In between such replicated runs, the system setup is re-initialised by clearing the main memory and eliminating effects of caches or buffers.

For a particular set of scheduling parameters, we have plotted the compute intensive programs in one graph and the rest in the other graph. The lower the value of our *fairness measure*, the fairer and more consistent the scheduler. Within each graph, we see that for a given CPU quanta, the *fairness measure* increases as the decay period increases. This is due to the fact that more time is required by the system to re-compute priorities. With smaller decay periods, the priorities of all processes are re-computed more often, and hence the priorities more accurately reflect the nature of the processes. Across the

graphs, as the decay period remains constant and the quantum size allocated increases, we see that the measure of fairness decreases. For example, take the case of decay period being equal to one second. As the quantum size is increased from 30 millisecs to 250 millisecs, the measure of fairness decreases from an average of 0.06 to 0.02. This is explained as follows: with a fixed decay period value, as the quanta value decreases, decaying occurs less frequently relative to the value of time quantum. The lower rate of priority re-computations results in unfairness. In general, the more frequent the priority re-computations, the more accurately is the behaviour of processes reflected, and hence the fairer the behaviour of the scheduler.

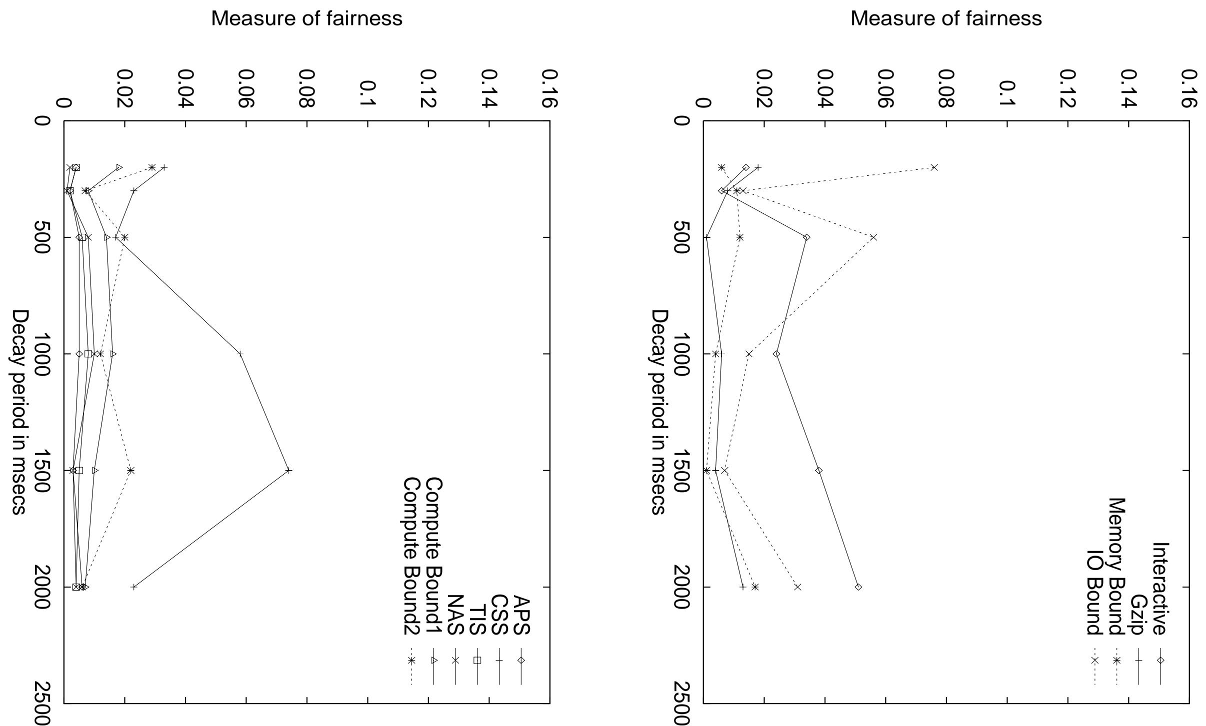
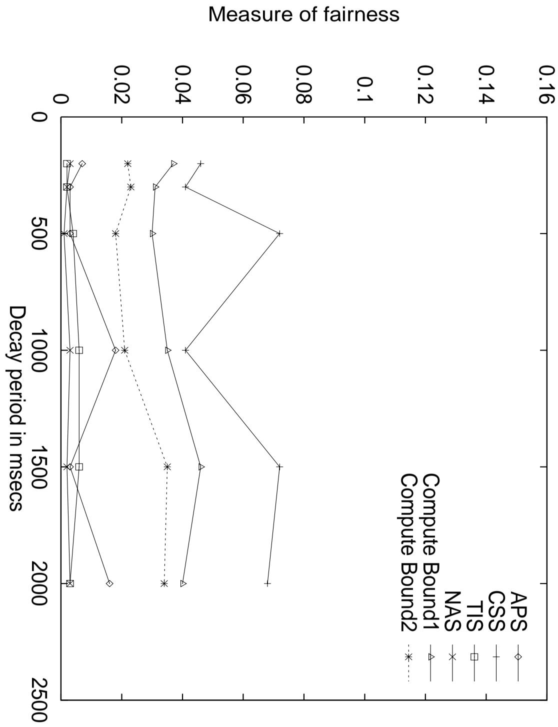


Figure 3.6: Decaying by 1.5, with CPU quanta of 100 millisecs.

Figures 3.6 and 3.7 show fairness results for decay factors of 1.5 and 4. Under the decay value of 4, decay is fast and hence the CPU consumed by an erring process which managed to occupy the CPU for quite an amount of time in the recent past, is quickly forgotten. That erring process is thus a competitor for CPU time on an equal footing with other processes. This is the reason why across different decay rates, with constant decay period and CPU quanta, the measure of fairness varies to the benefit of slower decaying,



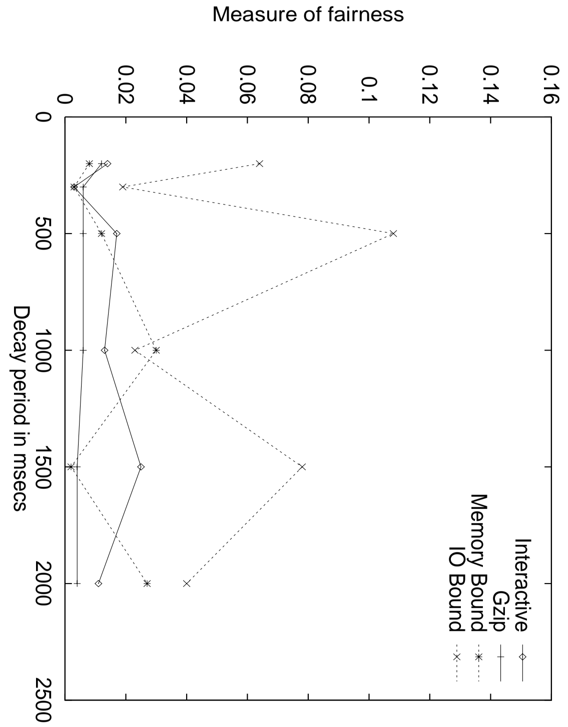
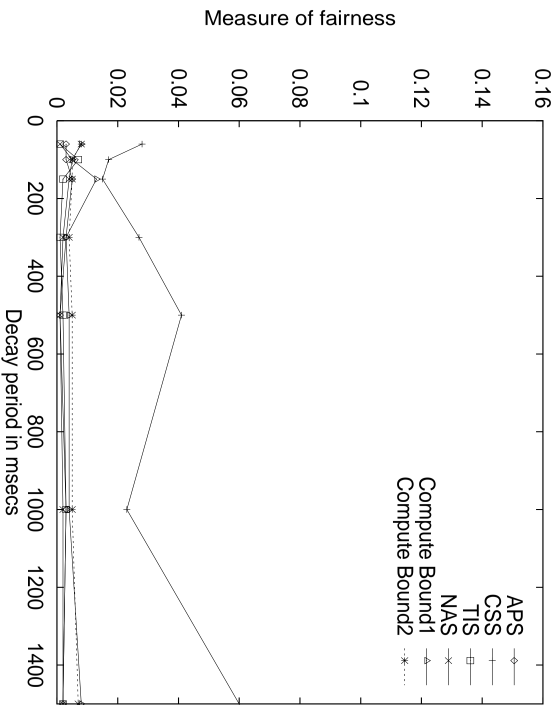


Figure 3.7: Decaying by 4, with CPU quanta of 100 millisecs.

as is evidenced by the graphs. The figures also illustrate that the decay factor of 1.5 is not only fairer but also gives a more consistent *fairness measure*, when the decay period is varied. Also, note that there are about seven compute intensive applications running; as a result, more time is spent in different levels of the priority rather than jumping across priority levels if the decay is slow. In fact, the *spread* value for decay by 1.5 is about seven.

Figures 3.8 to 3.10 show fairness results for a 4.3BSD type scheduler we built on top of the default scheduler on our Linux system. Two characteristics of the 4.3BSD scheduler contribute to the *fairness measure* in a significant manner. Firstly, recall that decaying is done based on the current load average of the system. The decay rate is thus slower if the load is higher and vice versa. This means that the *spread* value changes dynamically to the benefit of currently existing processes in the system. Secondly, re-computation of process priorities is done once every 4 ticks of CPU consumed, irrespective of the quanta allocated to a process. This means that every significant consumption of the CPU by any given process is immediately taken into account by the scheduling algorithm and the



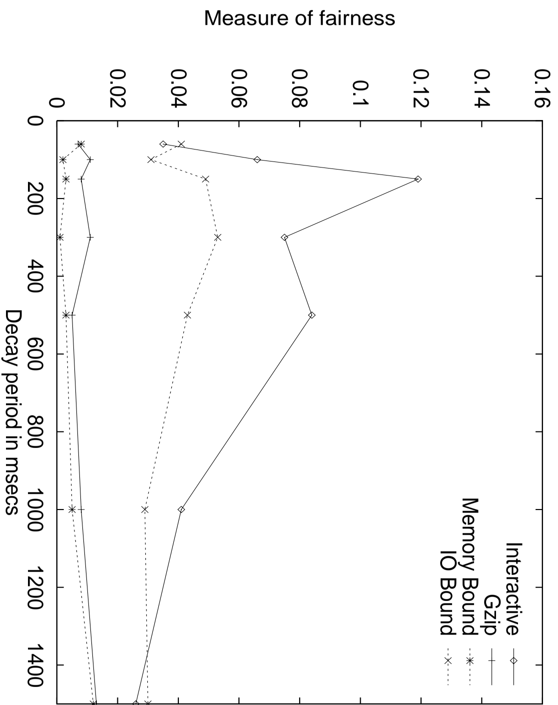
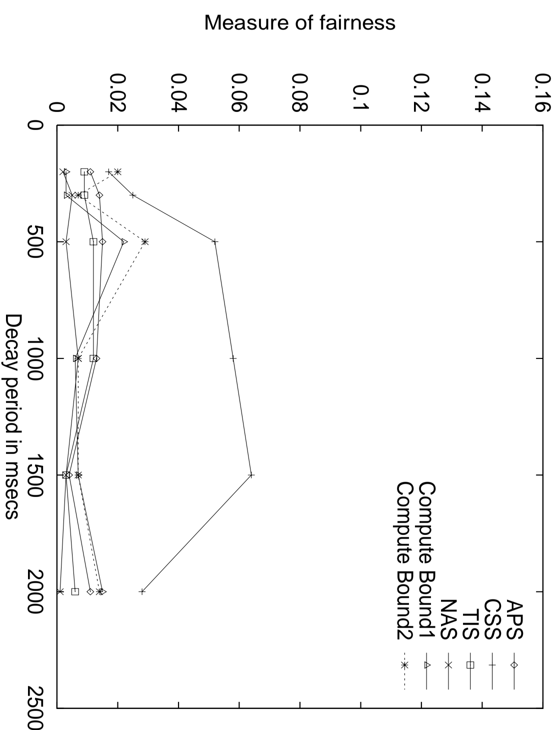


Figure 3.8: 4.3BSD Scheduler with CPU quanta of 30 millisecs.



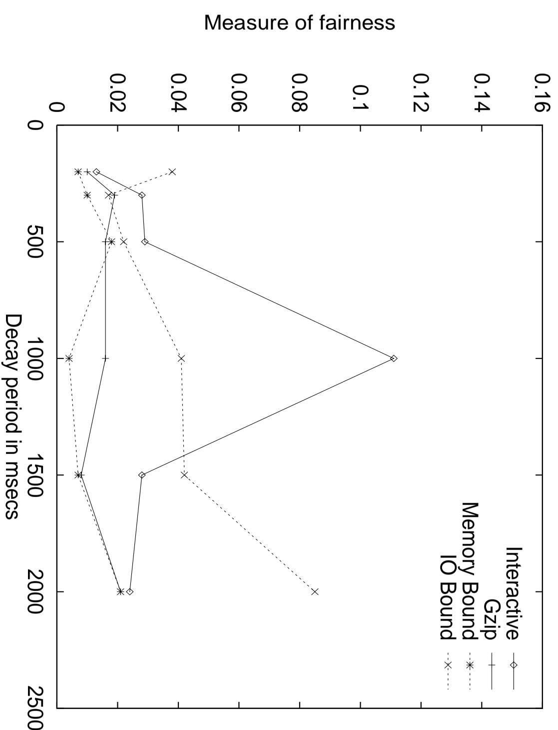


Figure 3.9: 4.3BSD Scheduler with CPU quanta of 100 millisecs.

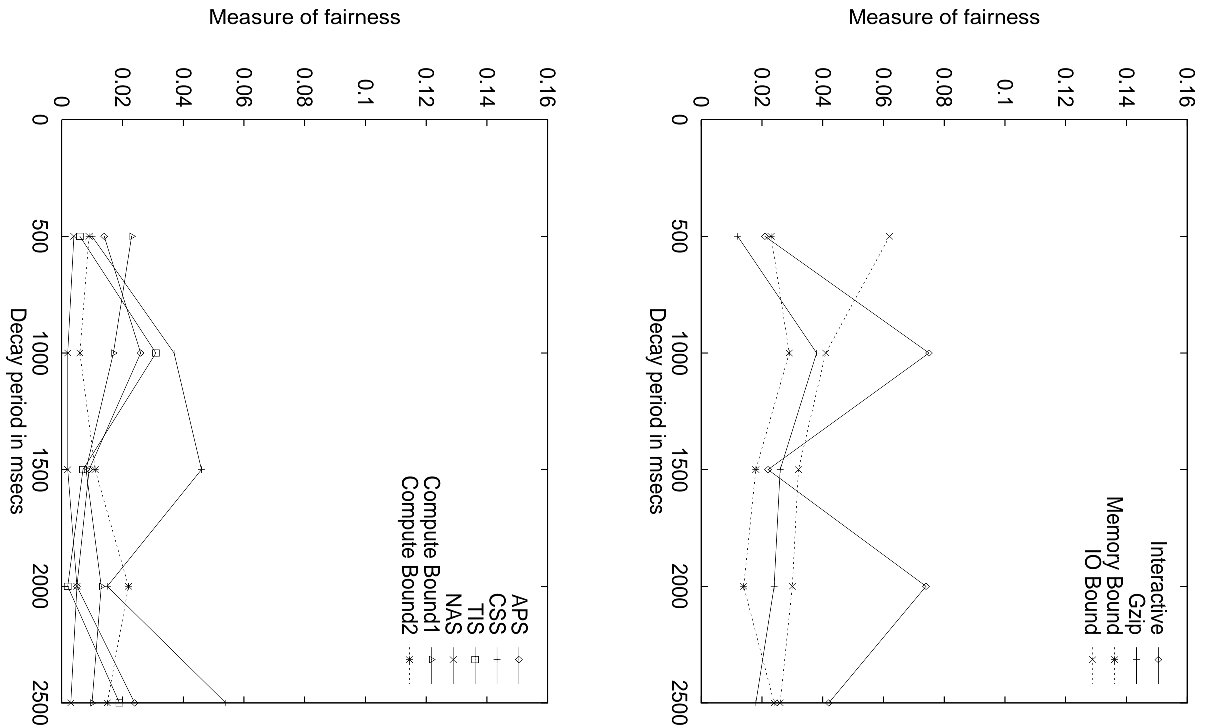


Figure 3.10: 4.3BSD Scheduler with CPU quanta of 250 millisecs.

priority is adjusted to reflect that. We note from the figures that as the quantum size increases, the *fairness measure* also increases, indicating that larger quanta per process is inherently unfair. This contradictory behaviour of 4.3BSD as compared to Bach scheduler is further explained in section 3.7 of this chapter. However, for all values of the scheduling parameters, we observe that the measure of fairness is always lower under the 4.3BSD scheduler than for the Bach scheduler with similar parameter values. Note that for a CPU quanta of 30 millisecs, the measure of fairness is close to zero for most of the programs. This may not be a feasible quantum size, as the context switch overhead would be prohibitively high.

To assess the effect of priority re-computation (done by the 4.3BSD scheduler once every 4 ticks), and to improve the *fairness measure* for a normal Unix system with Bach scheduler, we next studied the merits of re-computing the priority for each process as soon as its quantum gets over. This involves additional overhead in the context switch time, as the priority is re-computed and assigned to the switched out process at that time. But, as seen in Figure 3.11, the *fairness measure* has improved considerably in comparison

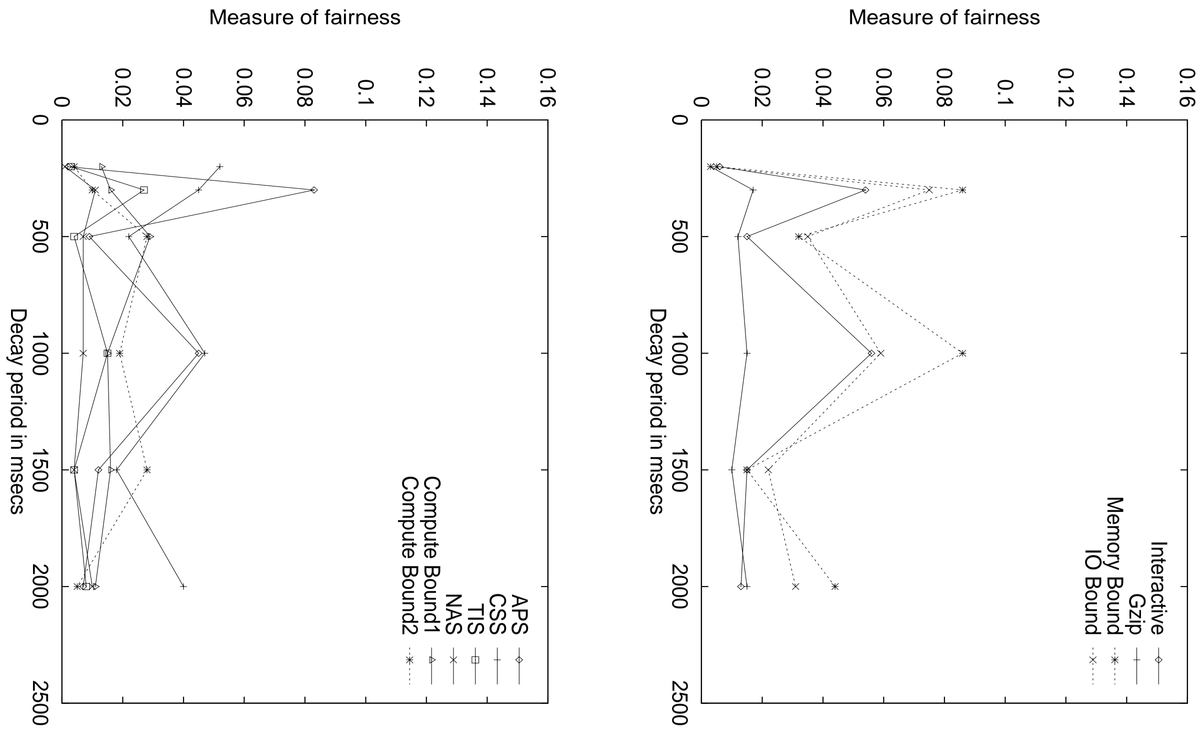


Figure 3.11: Priority re-computaion with CPU quanta of 100 millisecs.

with Figure 3.3. As we said earlier, the improvement is because of the frequent priority re-computations reflecting the process behaviour more accurately. This, in turn, results in the process scheduler being fairer than it was previously.

3.5 Experiments with compute bound processes

Fairness can not be assessed completely by considering only program elapsed times. The amount of CPU time that has been assigned to a program at various points in time during its execution should also be considered; i.e., the distribution of the CPU time that it receives over its real execution time. We therefore next studied the accumulated CPU time of each of a set of similar processes at various points during their concurrent execution. One might expect the accumulated CPU usage at these intervals to be more or less the same across all the processes.

In each experiment, we started n executions of the same compute bound program and measured their CPU usages once every second over a duration of 500 seconds. The value

of n was varied from 2 to 10 processes. The experiment was conducted on three systems - a SUN Sparc20 running SunOS 5.4 (using the System V Release 4 process scheduler), an Intel 486 running a Bach scheduler (Linux based), and an Intel 486 running a 4.3BSD scheduler (Linux based). For both the 4.3BSD and the Bach scheduler, the CPU quanta was 100 millisecs and the decaying cum priority re-computation was done once every second. We used compute bound processes to avoid inherent mismatches present across architectures, like different computing power, I/O rates and main memory availability. Thus, making comparisons across machines is justified as far as the process scheduler is concerned.

In order to monitor the CPU usages of the processes, we required some method of accessing the per process *proc* structure which contains such information. In the SUN Sparc20, the data structure used for allocating the proc structure to a process is not static, like an array, but done through dynamic memory allocation. Hence, directly accessing the proc structure in the kernel memory starting from the first proc structure and sequentially scanning through structures until the required process' proc structure is reached does not work. The system provides access to the kernel memory through a set of system calls which are part of the Kernel Virtual Memory (KVM) library. *kvm_open* opens the kernel and supplies the identifier which is later used in other *kvm* system calls. We used the *kvm_getproc* call to access the proc structures of processes. But, there was a drawback in this approach - the data read were cached by the system, resulting in the values not being up-to-date in successive calls. However, every-time a *kvm_open* was done the values got were up-to-date. This suggested that we had to call *kvm_open* before every observation point during our experiments. We measured the overhead involved for a single *kvm_open* call and found it to be 180 millisecs. Considering that the observations are done once in every second, the overhead involved is too high. Instead we used the alternate approach of accessing the */proc* device using *ioctl* system call. Fortunately, the overhead involved here was only around 10 millisecs and the values reported were accurate and up-to-date. For the other two process schedulers (Bach and 4.3BSD) the CPU usages of the processes were monitored using a system call that we added, *GetUsage*. This call takes as parameter

the pid of the process to be traced and returns the current CPU usage of the process.

In relation to the past work done on Fair Share Scedulers, where a share of the CPU is guaranteed to a particular group of processes, we now look into the share of the CPU that a compute bound process gets during the course of it's execution. If a scheduler is fair to all processes, the amount of CPU time consumed by each process in the system should be identical at every instant of measurement. We calculate the standard deviation of these measured values at every instant, over the range of n, i.e., 2 to 10. To nullify the effects of the different number of processes in the system, we divide the standard deviation by the average value. The scheduler for which these normalised standard deviation values are zero over the whole range of n will be considered to be the *fairest* scheduler.

Thus, our measure of fairness $\mathcal{F}$, at a given time instant is

$$\frac{\sigma\ (CPU_usage_i)}{(\sum_{i=1}^{n} CPU_usage_i)\ /\ n}$$

where,

CPU_usage_i is the CPU consumed so far by process i at that instant.

Before we look at the results of these experiments, consider graphically the effect of a *perfectly fair scheduler*. Figure 3.12 shows the expected CPU usage for a process on the z *axis*, given that there are n similar processes present in the system simultaneously, with n shown on the y *axis*, while the time elapsed is plotted in the x *axis*. The surface plotted is basically z=(x/y), since a *perfectly fair scheduler* is supposed to ensure that the CPU is shared evenly among the existing processes at any instant.

3.5.1 Results

Figure 3.13 shows the CPU usage measured on a SUN Sparc20 for a single process among n processes, with a magnified view in Figure 3.14. The upper figure shows the values for the first process that was forked off, while the lower one shows the values for the last process forked off, in each case. Note that the first process gets less CPU time than the last forked process. Further note that both the figures show significant deviations in CPU time

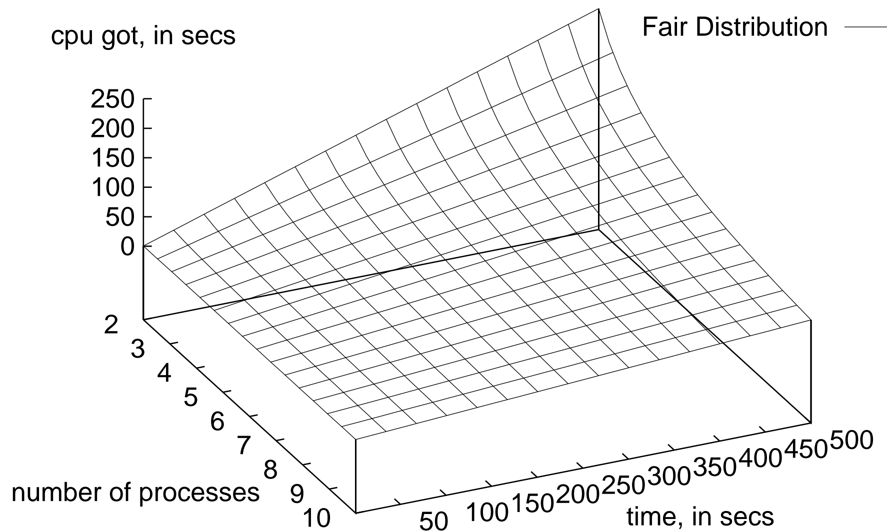
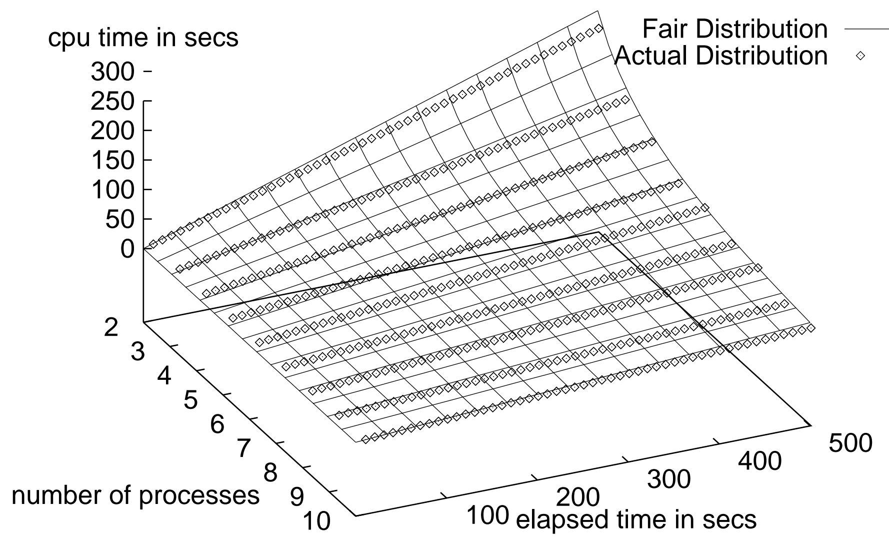


Figure 3.12: Expected CPU distribution in case of a Fair Scheduler.

distribution from the ideal case. These experiments were run under no-load conditions. As mentioned earlier, the SUN Sparc20's process scheduler is a System V Release 4 process scheduler. We see from these figures that for a single process, the effective CPU time assigned to it might be well below *or* well above the expected value. We also see from the graphs that when viewing the ideal distribution as a surface, then for both the actual cases plotted, the surface is either fully below or fully above it. For the case of two identical compute-intensive programs starting off simultaneously, the final CPU usage values after an observation period of 500 seconds are around 280 and 220 seconds. These values differ significantly from each other; recall that this was the outcome of using a process scheduler for which fairness was a stated design objective.

Figure 3.15 plots the values of our measure of fairness for the CPU consumed by the different processes present in the system in the above experiment. The top half the figure shows the plot for all odd number of processes while the bottom half shows the plot for even number of processes. To make the lines clear, every tenth value was used to plot the graph. During the first few (typically 50) seconds, there is considerable variation in the values of the measure, indicating that the system is yet to attain a steady state.



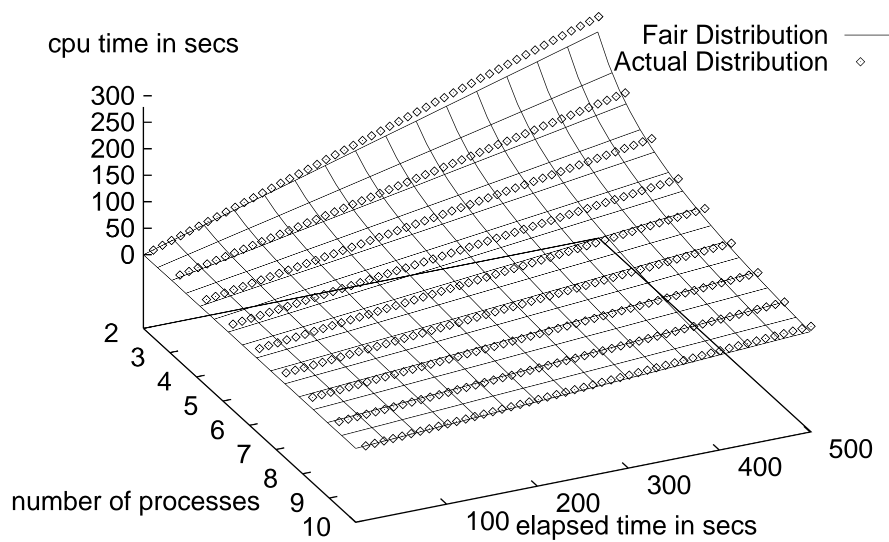
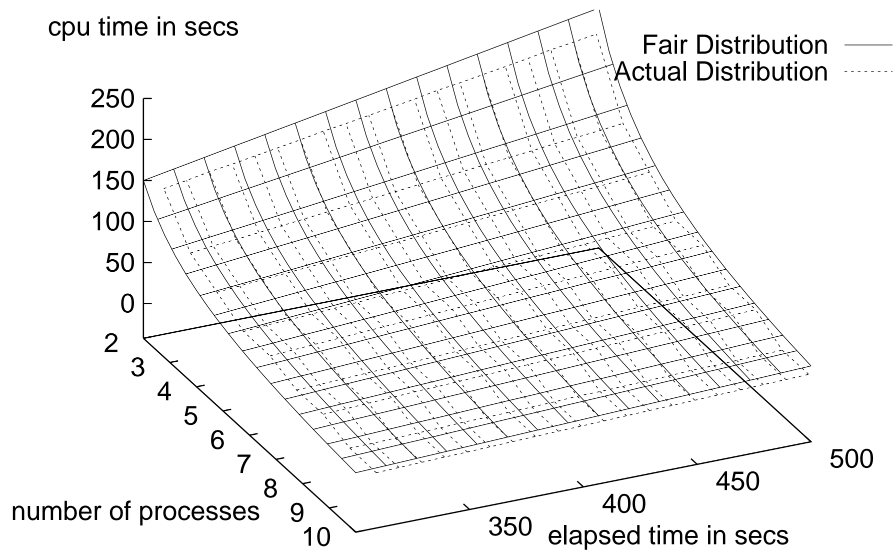


Figure 3.13: CPU time distribution: SUN Sparc20 Scheduler.



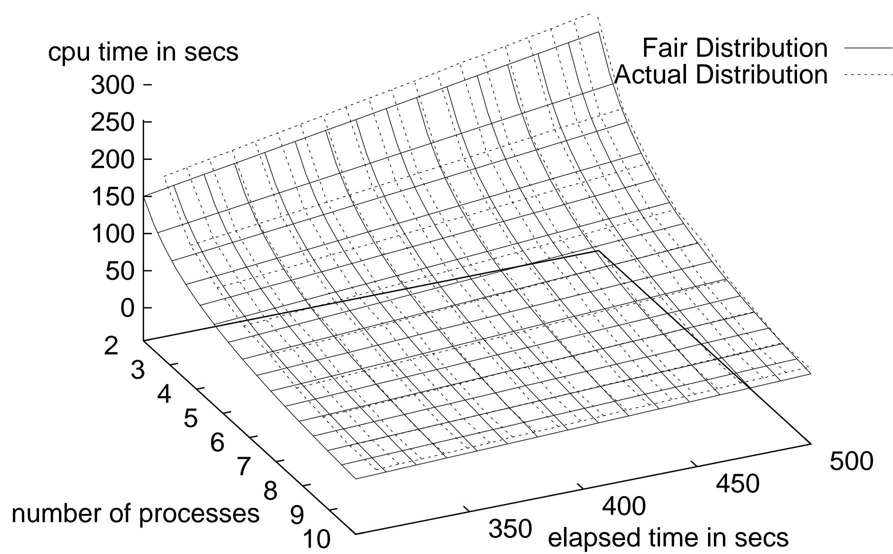


Figure 3.14: Magnified view of CPU time distribution: SUN Sparc20 Scheduler.

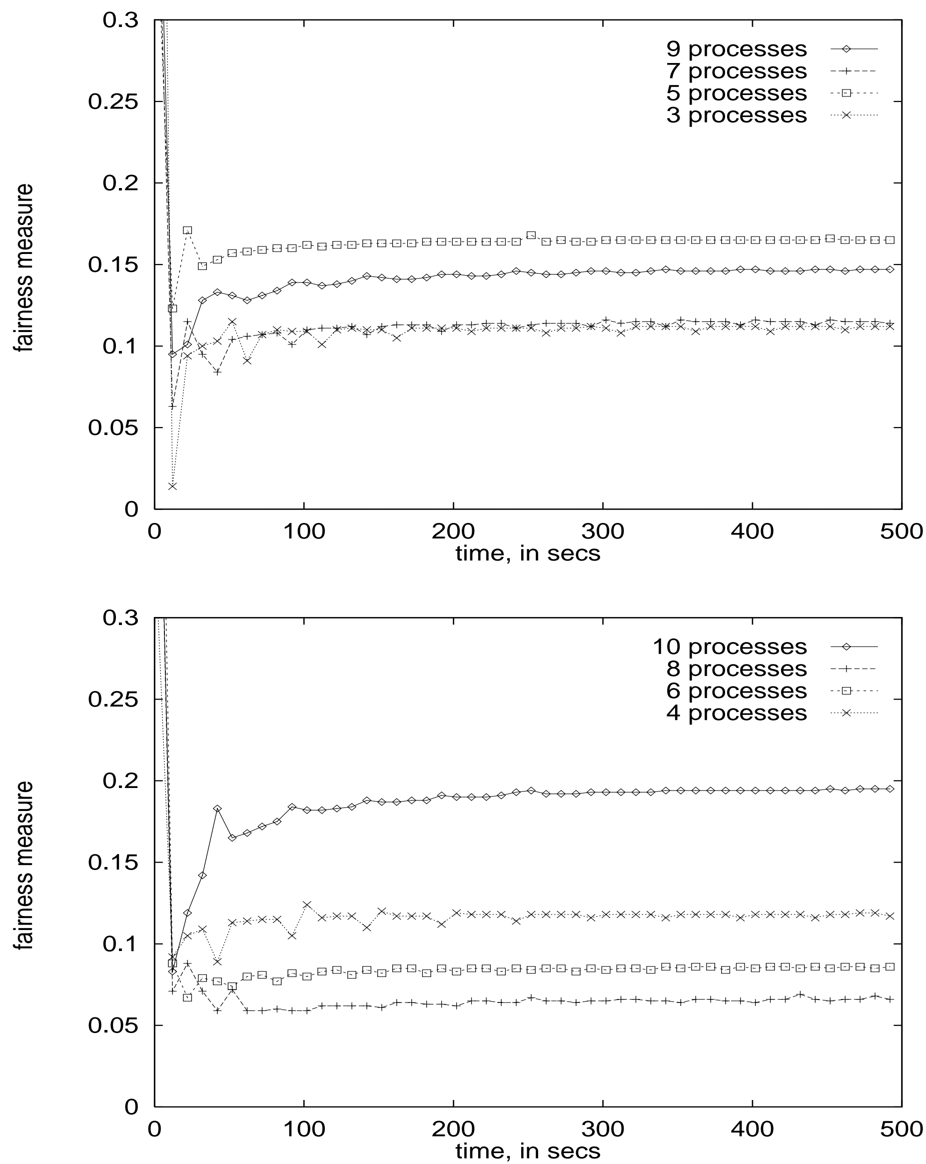


Figure 3.15: Fairness measure for SUN Sparc20 's System V Release 4 process scheduler.

Beyond that time period, the measure seems to stabilise to a constant value. Thus, even though the process scheduler performs poorly in terms of fairness to processes, it does so with admirable consistency. As we will see later, in addition to poorer performance, this consistency is missing in a Bach scheduler. We also see from the figures that as the number of processes present in the system increases, the *fairness measure* also increases in general. This means that the scheduler behaves less fairly when the load present in the system increases. Since any system might perform well under low load conditions, it is the behaviour under high load conditions that we put under greater scrutiny. Curiously, the best *fairness measure* values are attained by the system for the case of n equal to 8 and the next best at 6. For the rest of the values of n, the measure is above 0.1.

Figures 3.16 and 3.17 show the measures of fairness for a Bach scheduler under different number of processes present in the system. Observe that in most cases, the measure of fairness stabilises as time passes; there are exceptions where the value continously increases. Also, note that there is no steady value for the measure, neither within a particular experiment nor across experiments. Therefore, we have plotted results from three repetitions of each experiment. In each of these plots we see that the measure is different for the same value of n. The graphs also reveal that in general, as the number of processes present in the system increases, the *fairness measure* also increases. This means that the fairness performance is poor for larger values of n, which can be attributed to the fact that the *spread* value of a Bach scheduler is 5. Had the *spread* value been higher, the measure of fairness would be lower even for higher values of n. We will confirm this observation when we look at results for the Bach scheduler with a different decay rate.

In order to better appreciate this measure and its significance, we next looked at CPU usage times of various processes in an experiment with n = 6. In Figure 3.18, the upper most plot shows the CPU usage of six identical processes that started executing simultaneously under a Bach scheduler. The middle plot in the figure is a magnified view of results for the same experiment, showing the last 100 secs of the experiment. For this particular experiment, the measure of fairness starts off from a value of 0.3 and steadily decreases to 0.06, as seen in the lower most plot. Thus, at the end of the experiment,

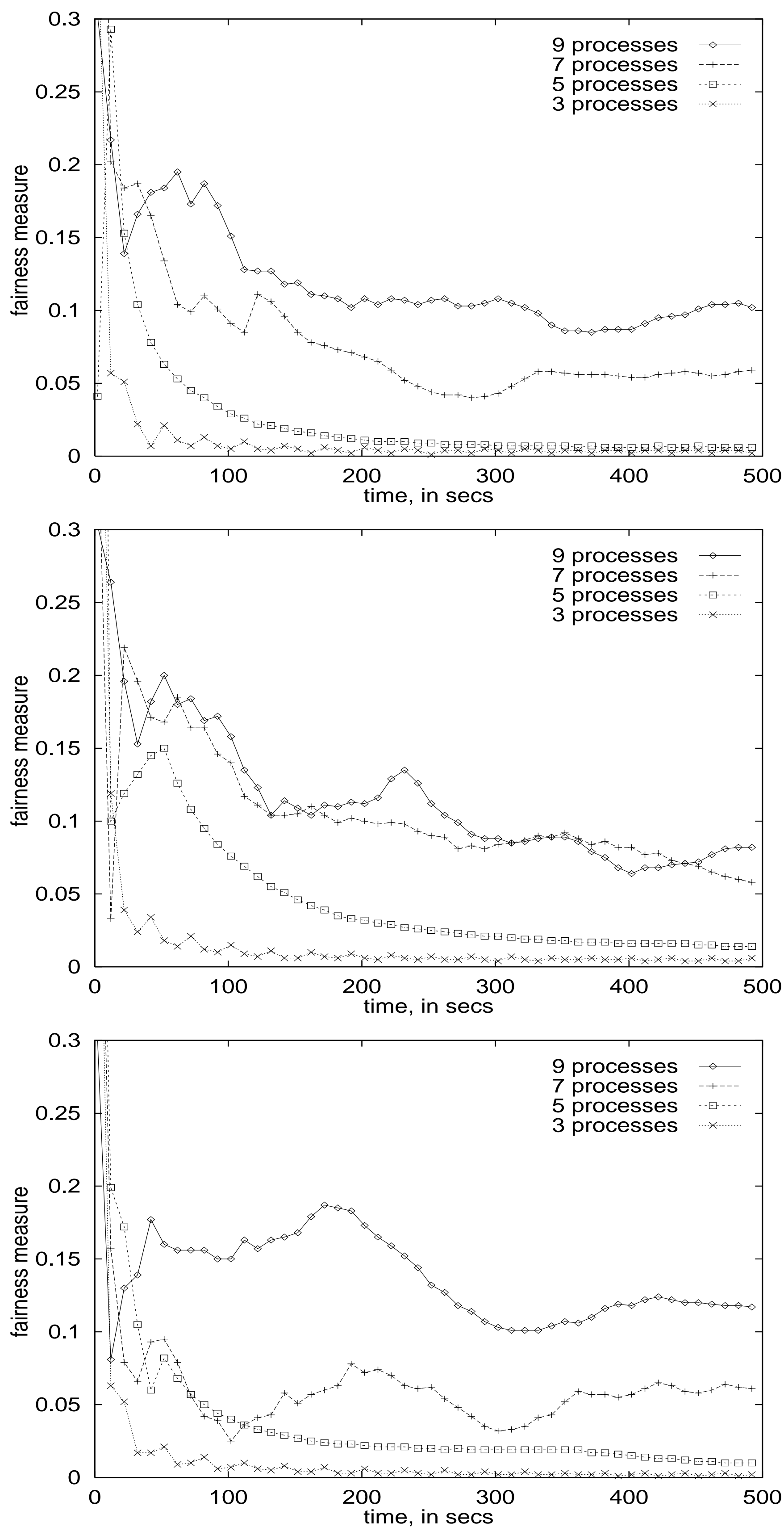


Figure 3.16: Fairness measure for odd number of processes: Bach scheduler.

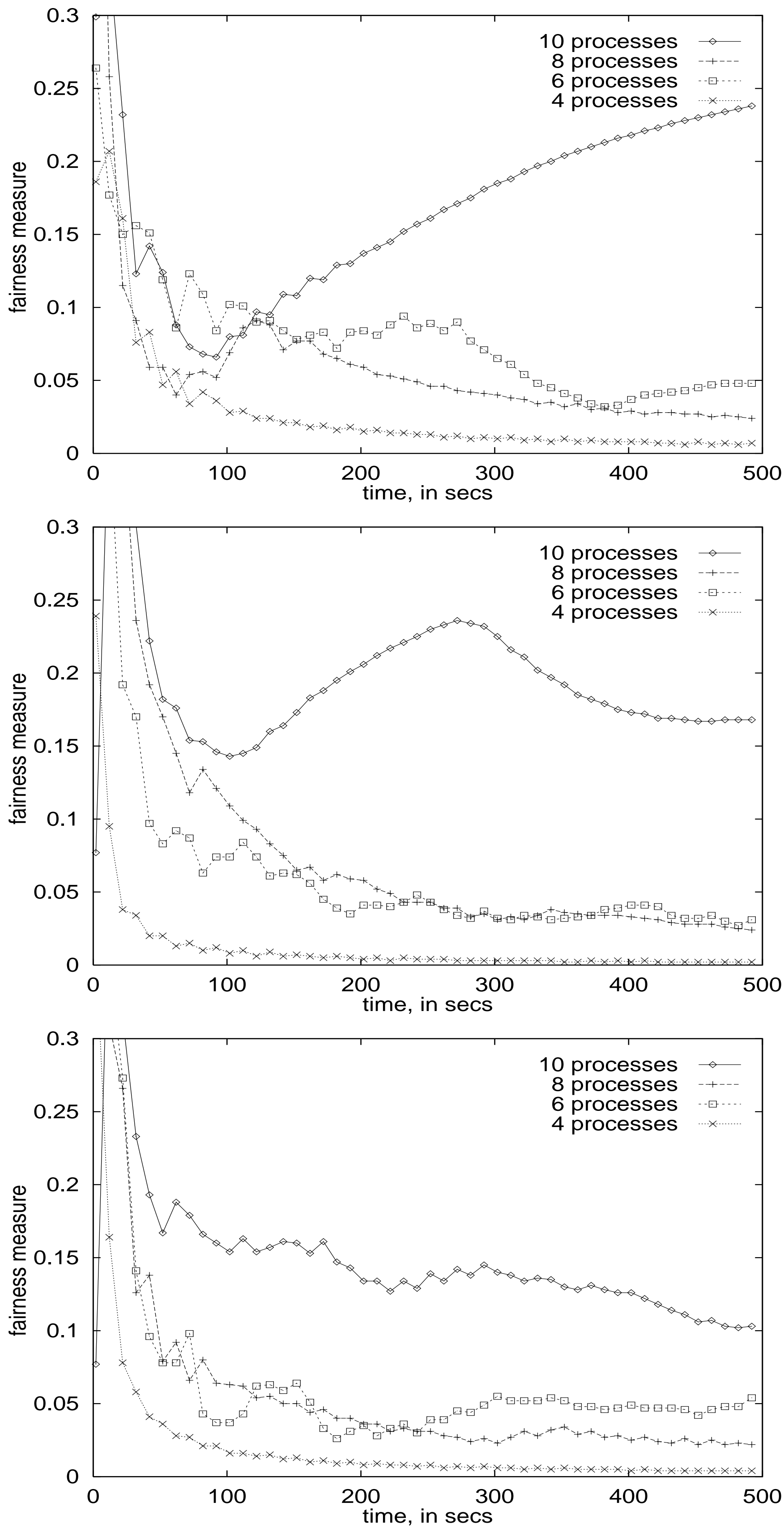


Figure 3.17: Fairness measure for even number of processes: Bach scheduler.

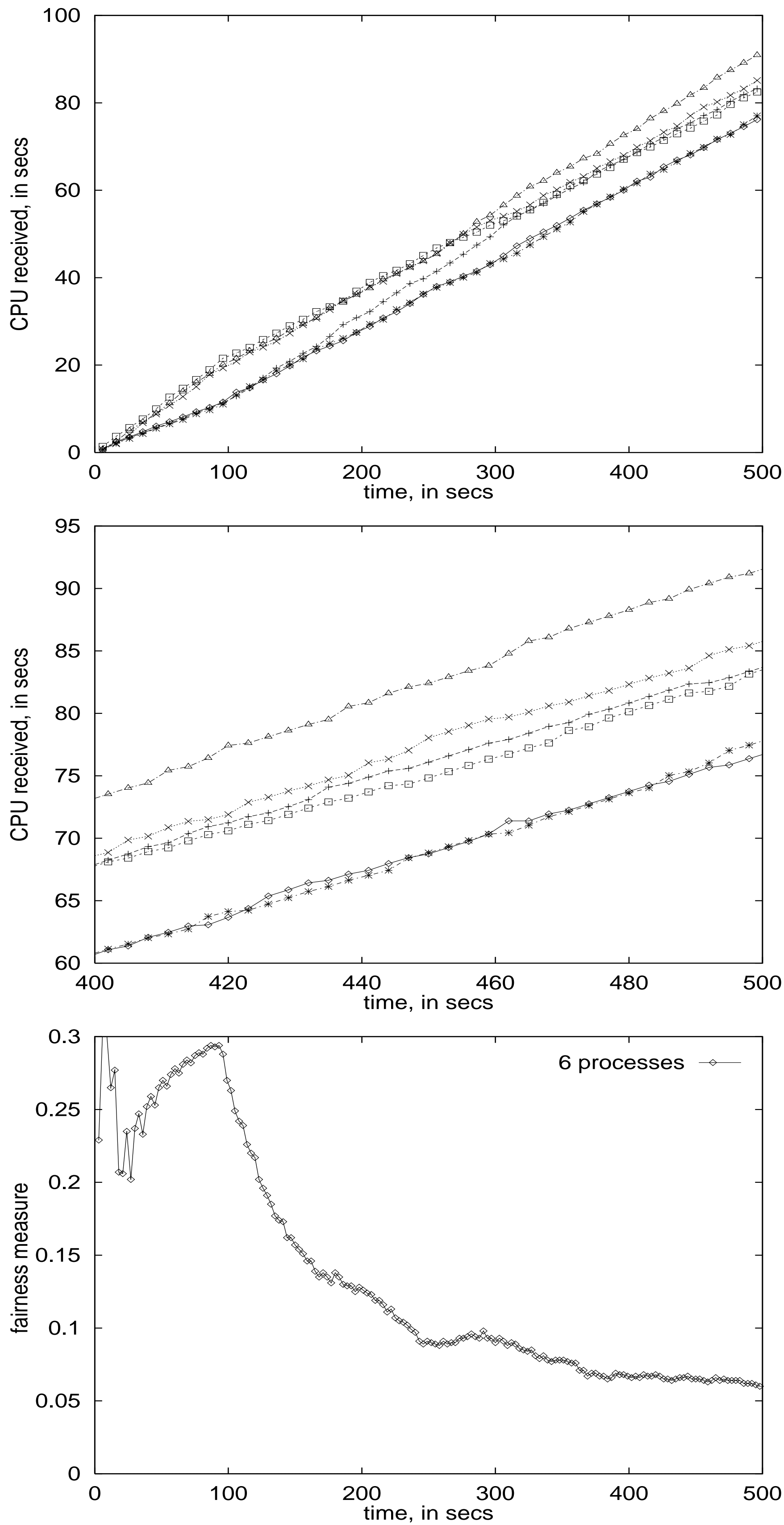


Figure 3.18: CPU usage for six processes and it's fairness measure: Bach scheduler

i.e., at 500 seconds past the start time, we have a *fairness measure* of 0.06, and the largest difference in CPU usage between any two processes is around 15 seconds. This difference is quite high, considering that if the scheduler had been fair, each process would have consumed only about 83 seconds of CPU time. Also, the first process, which has the highest CPU usage, had received the same amount of CPU (76 seconds) as the last process (lowest CPU usage) at the end of 500 seconds - almost 80 seconds prior to it. These observations emphasise the value of our *fairness measure* and its practical significance.

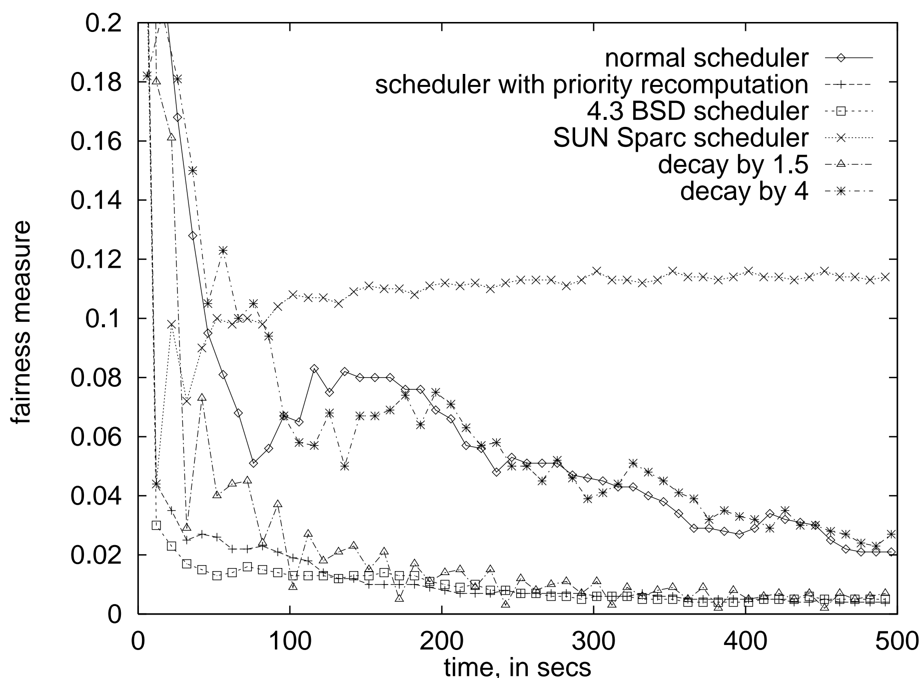


Figure 3.19: Scheduler Fairness: seven identical compute bound processes.

To compare all of the schedulers with respect to the *fairness measure* under consideration, Figure 3.19 was plotted for the case of seven identical processes starting off simultaneously. Observe that as the decay factor increases, the *fairness measure* improves, which also means that as the *spread* value increases, there is improvement in the fairness of a scheduler. The 4.3BSD scheduler and the scheduler with priority re-computation have consistently low values, which means that they are the best from the perspective of fairness. Also, the scheduler with a decay factor of 1.5 performs fairly close to these best cases, and better than the Bach scheduler, which has a decay factor of 2. This is due to the fact that if the decay rate is high (due to higher values of the decay factor), the value

of recent CPU usage decreases fast, resulting in the scheduler forgetting the heavy CPU usage of an erring process, if any. Thus, this erring process is back in contention for the CPU quicker than it would have been if the decay rate had been lower.

Note that the SUN Sparc20's System V Release 4 process scheduler performs the worst among the schedulers, with *fairness measure* of 0.1 which is very high as compared to others. But, as pointed out earlier, it performs consistently, unlike the Bach scheduler which sometimes performs well, but sometimes performs badly. This observation is confirmed by looking in to the Figure 3.16 for the case of seven identical processes. In these experiments, the CPU quanta size was fixed at 100 millisecs, and decaying cum priority re-computation was done once every second. The scheduler in which priority re-computation is done at the end of a quantum performs well, since recent behaviour of a process is captured by these re-computations. Thus, even if an erring process manages to get more CPU time, it is put into the proper priority level (low) by having it's priority re-computed.

Why does the 4.3BSD scheduler perform so well? To answer this question, we experimented with the scheduler in two ways. We repeated the above experiment of starting off seven identical processes simultaneously, for two special cases: (i) without priority re-computation being done once every 4 ticks (40 millisecs), and, (ii) without the load based decay factor that the scheduler uses. Figure 3.20 shows the results. Observe that priority re-computation plays a vital role as for as fairness is concerned. Without it, the measure is consistently higher, as seen from the upper plot in the figure. Note that the load based decay performs reasonably well even without priority re-computation, confirming our suspicion that the *spread* value plays a role in the fairness aspects of a process scheduler. As pointed out, the *spread* increases in the 4.3BSD scheduler as the load increases, resulting in the processes getting the CPU evenly.

3.6 The Phenomenon of Overtaking

The schedulers that we have studied are thus fair to varying degrees, in terms of the evenness with which CPU time is distributed among competing processes. Consider next the following fairness related question - Is it possible that a process starting off after

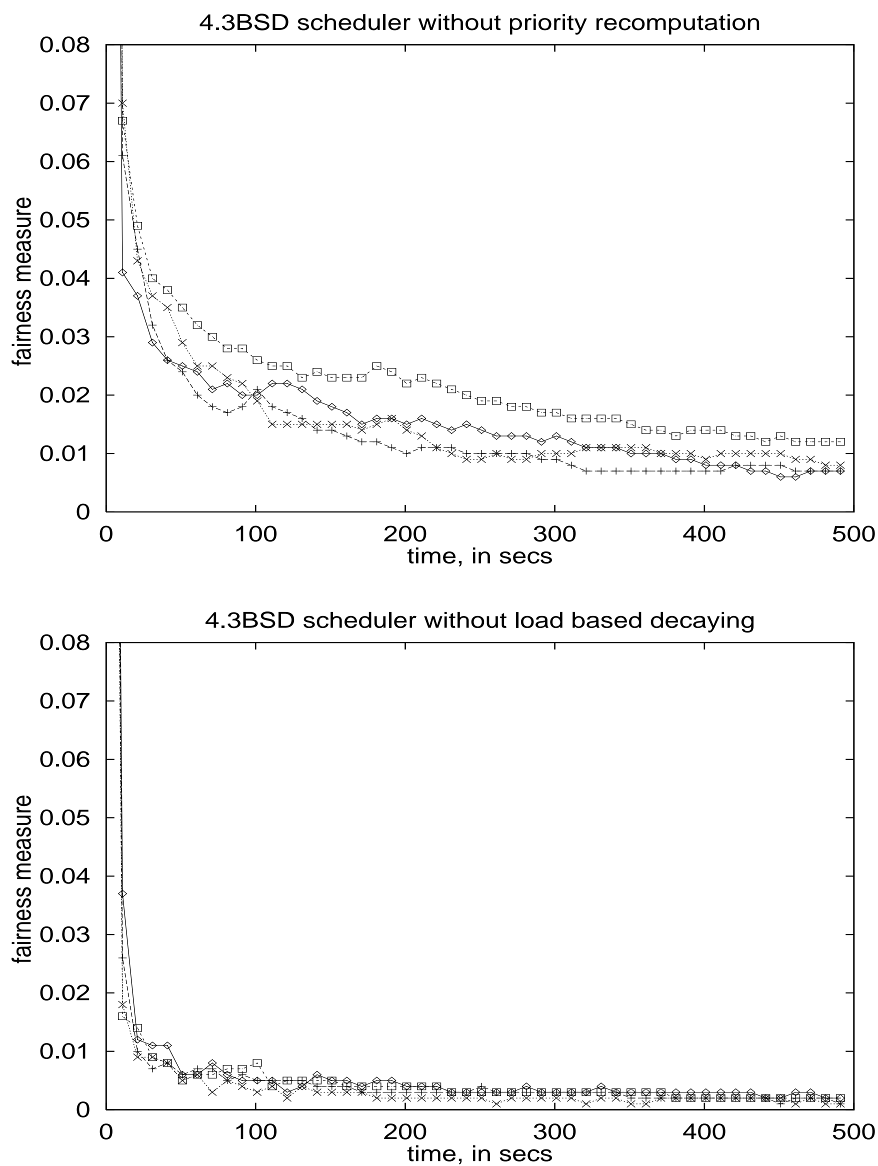


Figure 3.20: Fairness measure for 4.3BSD process scheduler: seven identical compute bound processes.

another identical process can overtake it and finish before that older process? In our next experiment, we started off n identical processes simultaneously, and followed this with two more identical processes after an elapsed time of 30 seconds. The CPU usages of all the processes were observed for a period of 500 seconds.

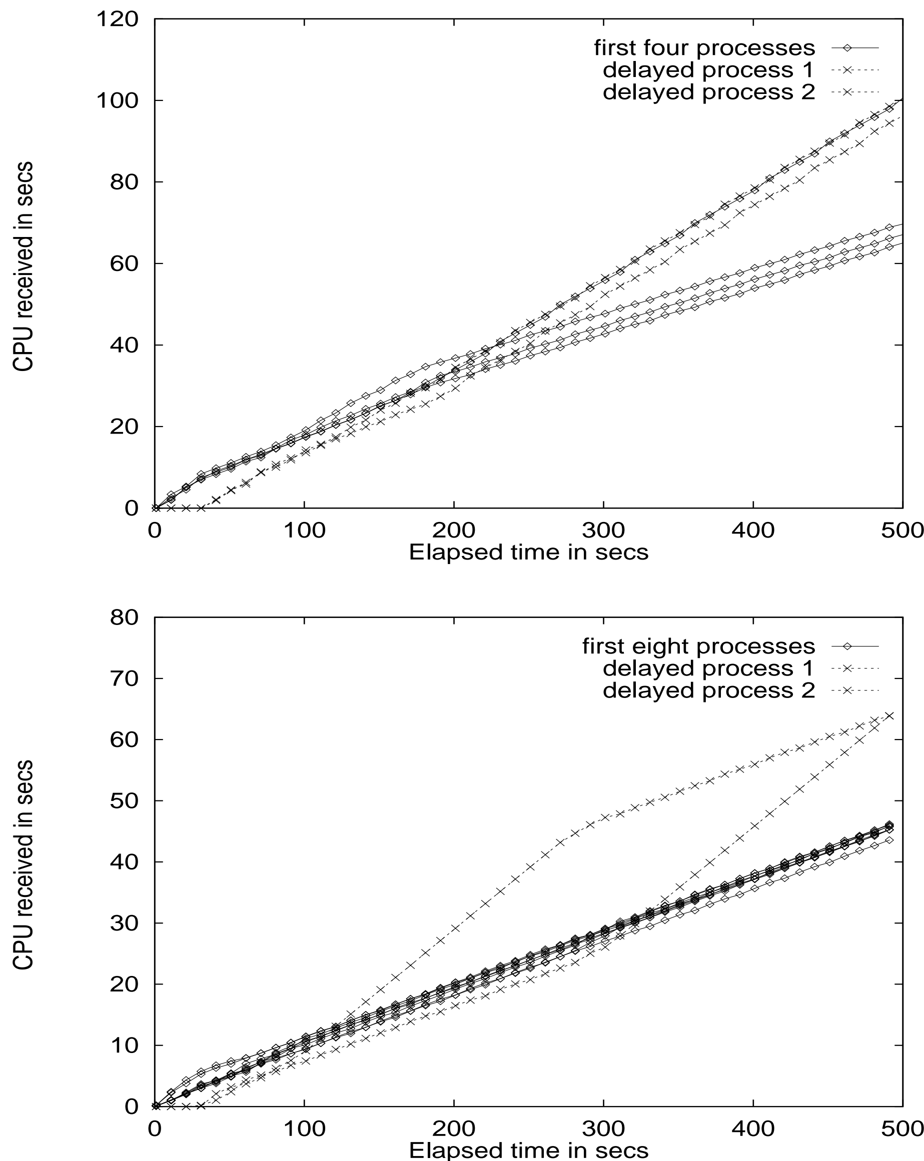


Figure 3.21: Overtaking: Bach scheduler.

Figure 3.21 show results for n equal to 4 and 8 for a Bach scheduler. Observe that overtaking does occur. We explain this phenomenon as follows: Even if all the processes are forked off *simultaneously*, not all of them receive CPU time equally during the first few millisecs. Further, when decaying of recent CPU usage is done, the inequality is further

compounded. So, there is a distinct possibility that one process is alone in its priority level for its entire lifetime. As time elapses, this process moves down the priority levels, i.e., it gains higher priority, and gets hold of the CPU. But, since it is alone in this highest priority level at this point in time, it continues to receive CPU quanta until it moves on to a different priority level. This happens since it's priority is re-computed based on recent CPU usage only at the end of the current decay period.

Considering that the spread value for a Bach scheduler is 3, this erring process is back in contention for the CPU after the next three decay periods, irrespective of the number of processes present in the system. So, it occupies the CPU once again for the whole of the decay period, and so on. This process could be one of the delayed processes that were started off after a delay of 30 seconds. Thus, process overtaking occurs.

Figure 3.22 supports this explanation. In this experiment, ten identical processes were started off simultaneously on a Bach scheduler and their CPU usages were monitored for a period of 500 seconds. It is clear from the graphs that a particular process is being treated with undue favour. We observed that the 'breaking away from the pack' can occur at any time during the course of the run - sometimes close to the start of the experiment, sometimes as late as halfway through the 500 second experiment.

Figure 3.23 illustrates the phenomenon of overtaking on a SUN Sparc20 system. The figure plots the values of CPU usage for n value equal to 2 and 6. Here, overtaking occurs because re-computation of priority (based on the recent CPU usage, and hence the proper status of a process, whether it is starving for CPU time or occupying the CPU excessively) was not done frequently enough. As we see from Figure 3.24 and Figure 3.25 showing experimental results with the 4.3BSD scheduler, if priority re-computations are done more frequently, the phenomenon does not occur. These two figures not only emphasize the importance of priority re-computation but also the validity of the *fairness measure*, as seen by the low values of the measure for the 4.3BSD scheduler and the Bach scheduler with priority re-computation. The lower the measure the fairer the scheduler, resulting in no overtaking of processes under any load conditions.

We confirmed our explanation of process overtaking through simulation. We simulated

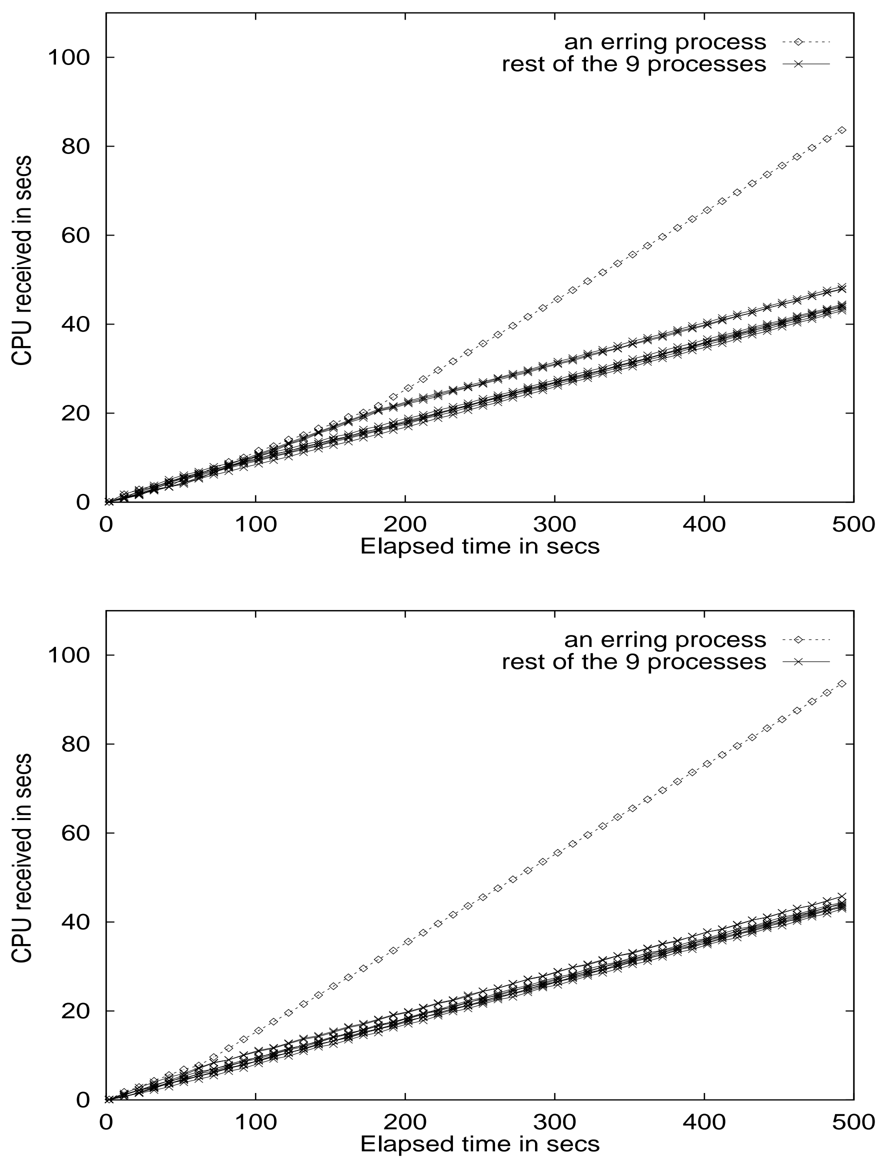


Figure 3.22: An erring process among the ten processes that started simultaneously.

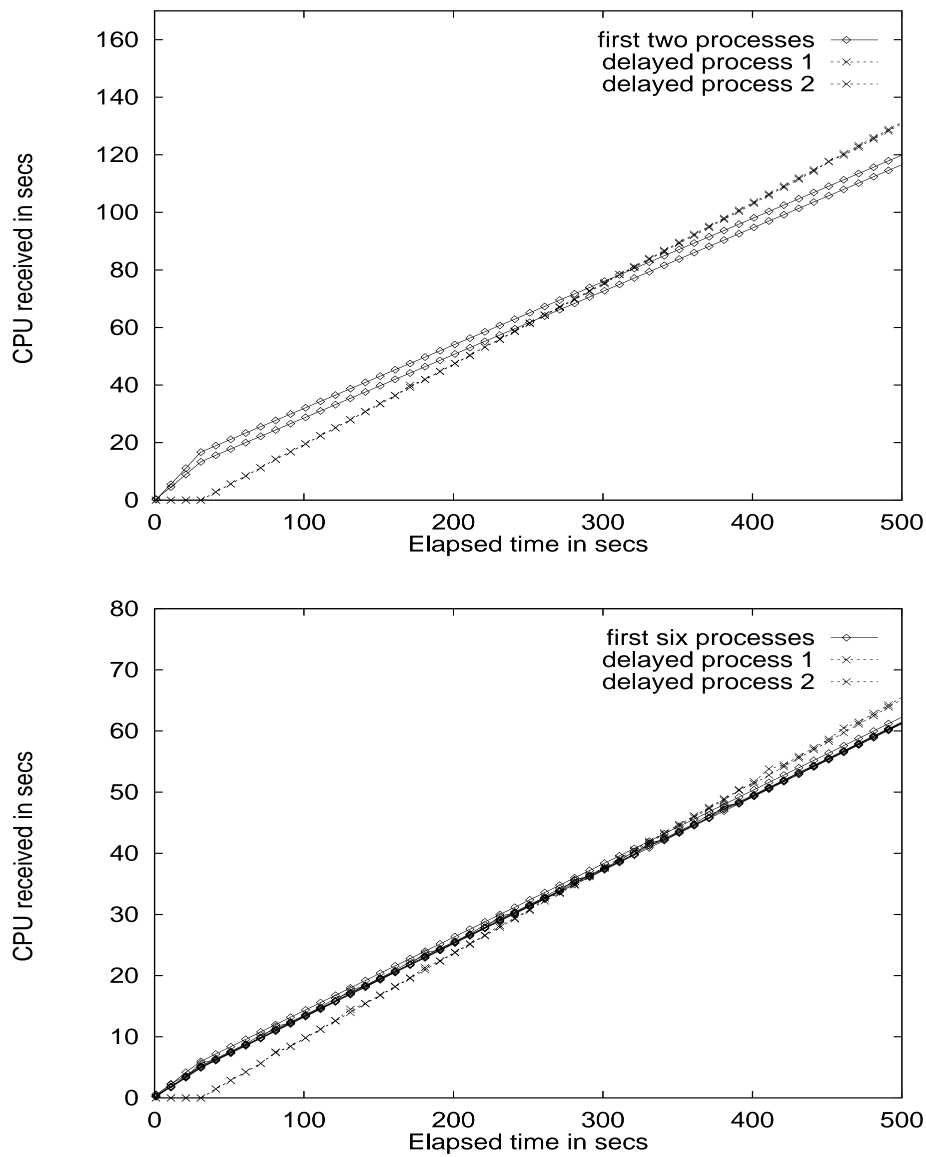


Figure 3.23: Overtaking: SUN Sparc20 process scheduler.

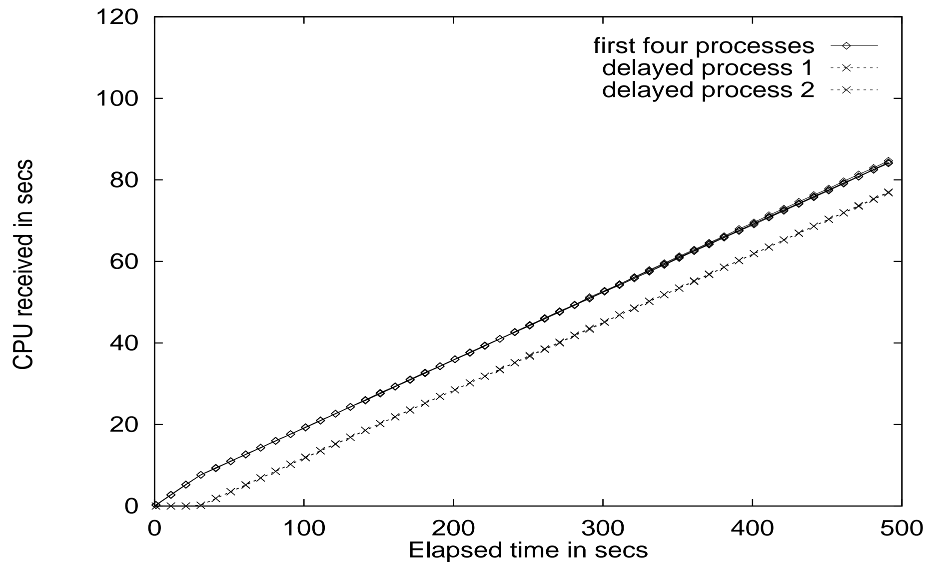


Figure 3.24: No overtaking: 4.3BSD process scheduler.

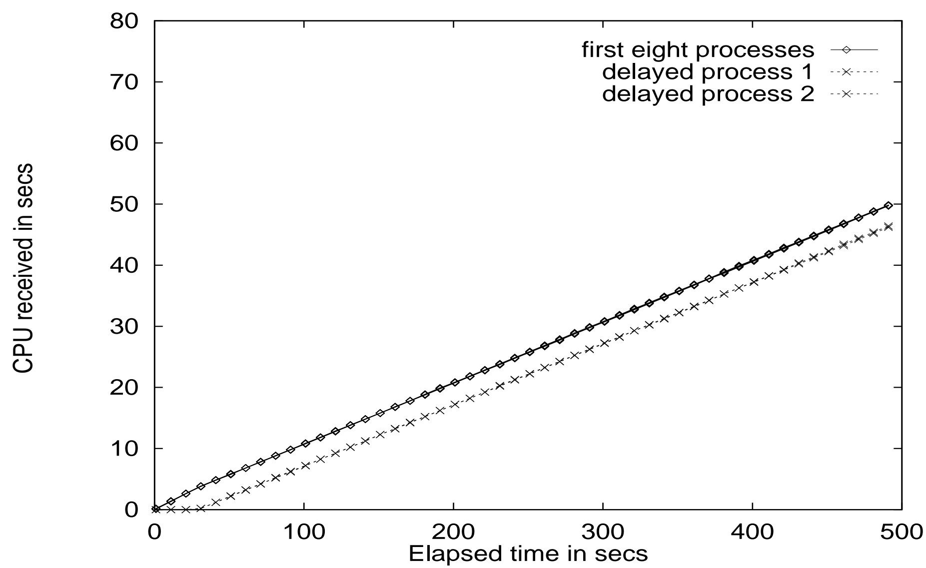


Figure 3.25: No overtaking: Bach scheduler with priority re-computation.

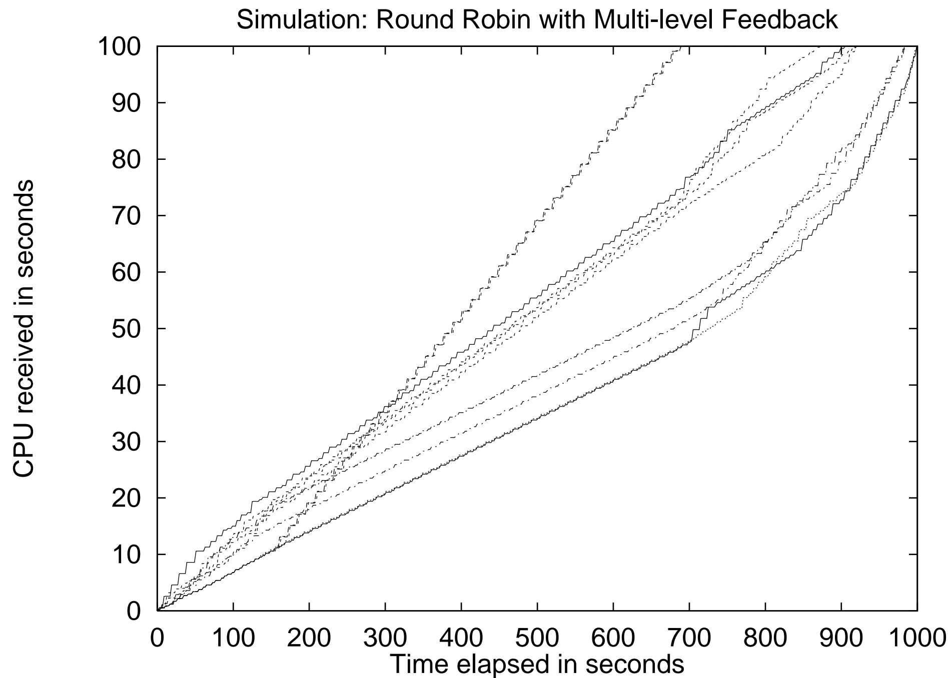


Figure 3.26: Simulated Bach scheduler for the case of ten processes.

the Bach scheduler, and studied the effects of varying numbers of processes being forked off. The start of the first decay period was also varied with respect to the start time of the execution of the processes. We observed that for different decay period starting times, the fairness of the scheduler also varied - from being fair to all processes, to being absolutely unfair to some of the processes. Figure 3.26 shows a simulated instance of the scheduler being unfair. Here, we have 10 identical processes each with an execution time of 100 seconds starting off simultaneously. Note that two of the processes have gotten away from the rest and managed to complete their execution more than 300 seconds ahead of the last finishing process.

3.7 Further Comments

For a Bach scheduler, the priorities are re-computed once every decay period. Only when that happens, do the priorities of all processes correctly reflect the current nature of the processes. So the best case for fairness in Bach scheduler is when the cpu quanta allocated per process is X msecs and the decay period is also X msecs. Instead, if they are X msecs

and 10X msecs respectively, the scheduler is not up-to-date with the current CPU usages of processes. It should be noted that in 4.3BSD, the priority of a process is re-computed every 4 units of CPU received by it, irrespective of the decay period. Thus the fairness measure truly indicates the fairness of a particular process scheduler according to it's underlying nature.

3.8 Conclusion

In this chapter we studied the fairness of process schedulers through two types of experimentation. Both lead us to the conclusion that among the schedulers studied, the 4.3BSD process scheduler performs the best in terms of providing *fair* treatment to the processes present in the system. We attribute this to the frequent priority re-computations done by this scheduler, and therefore introduced a variation to the Bach process scheduler, where priority is re-computed for every process as soon as its alloted quantum is over. In general, this resulted in improvement of the Bach scheduler's fairness measures.

We observed that SUN Sparc20 process scheduler treats processes in an unfair manner, allowing a young process to overtake and finish ahead of identical older processes, despite the young process' late start in life. This unfair overtaking problem can be corrected and the scheduler improved upon, since the set of parameter values under which the scheduler operates is configurable. However, this may not be a simple task, as there is no documented correlation between the scheduler table values and the standard scheduling parameters.

One of the assumptions made in past analytical evaluations of process schedulers is that the decay period and the evaluation time period are aligned. We saw that this is not a good assumption; during the course of our experiments, we had varying values for the measure of fairness and CPU usages of processes, despite keeping scheduler parameters and the experimental setup fixed - this was found to be due to the actual start time of the decay period calculation.

Chapter 4

Soft Real-Time Scheduling

4.1 Introduction

Traditionally, real-time systems used dedicated real-time operating systems (RTOS) with the necessary support for guaranteed execution times. Today, this notion of dedicated systems is changing, and there is interest in developing real-time applications on general purpose operating systems (GPOS). The reasons for this change are many - the widespread availability of GPOSs, portability of programs written over a GPOS as opposed to a RTOS, and the ease of code development due to the large set of developmental tools available over a GPOS - which result in shorter development time and lower costs.

In this chapter, we first look into the work done in the area of soft real-time systems built on general purpose operating systems, followed by a description of the scheduling algorithms implemented. We then describe the modifications we made to the Linux kernel for our experiments in using decay usage schedulers in soft real-time situations, and report on the results of these studies.

4.2 Background

A large body of work has been done on real-time scheduling in various environments, including processor, I/O and transaction scheduling. Most of this assumes an infinite

range of priority values and a custom scheduler on a real-time system. The suitability of Unix for real-time applications has been investigated by some researchers. Furht [7] identifies *performance and determinism* as the essential properties of an operating system to support real-time applications, and goes on to show that REAL/IX, a fully preemptive Unix based system compares favourably to a special purpose real-time OS. Mizuhashi and Teramoto [15] come to the same conclusion based on another real-time Unix, RX-UX 832. Wells studies two other GPOSs, SCO XENIX System V and OS/2 [18], concluding that both are viable for real-time applications, with OS/2 being particularly well suited due to it's high predictability. Wainer [17] implements facilities for real-time scheduling under MINIX and evaluates the performance of two real-time scheduling algorithms on real workloads.

While these studies demonstrate that a GPOS can be used to support real-time applications, none of them addresses the issue of priority assignment to the real-time tasks. This problem is addressed by Adelberg *et al* [1], [2]. They propose three strategies for priority assignment in a traditional multi-tasking environment, and evaluate them through simulation. Our work involves an implementation of these priority assignment algorithms on top of a Unix operating system and their evaluation through experimentation. We also introduce two new performance measures to describe the effect of these algorithms, apart from the usual measure of *miss ratio*.

4.2.1 Priority Assignment

For scheduling purposes, GPOSs assign priority values to processes; the process with the highest priority is assigned the CPU. These priority values are discrete and lie in a fixed range, unlike the continuous range of priorities associated with real-time scheduling algorithms like Earliest Deadline (ED) and Least Slack (LS). Thus, the emulation of these algorithms on a GPOS requires an appropriate priority assignment algorithm.

The essential idea here is to assign the highest priorities to the tasks with earliest deadline or least slack. This poses a problem in a GPOS with a limited number of priority levels. Since the levels are limited, when a real-time task arrives in an empty

system, what priority should it be assigned? If it is set high and many tasks arrive with earlier deadlines, the scheduler will run out of priority levels. A similar problem occurs if the priority is set too low. Setting the priority to a middle value does not help much either. Whenever a task arrives, if it is assigned the middle value as priority, the priority range must grow *exponentially* for the priority assignments to be according to the earliest deadline first or least slack first algorithm. Thus, to guarantee correct priority assignments to 6 tasks, we require 64 priority values.

Our experiments were done with two versions of Unix scheduler, the traditional Bach one and the POSIX Unix. While the traditional Unix has 40 priority levels for the user processes, there are 128 priority levels in POSIX Unix specially designated for real-time tasks. The priority of a process increases when it goes to numerically lower priority levels in a traditional Unix system. In POSIX Unix the priority decreases when the process assumes a lower priority level.

Let n refer to the total number of priority levels in the system. In order that the discrete and fixed set of priority levels available in the system be used efficiently, the following function is used for mapping the real number (R) obtained from the algorithm to one of the priority levels available in the system.

$map_{lin}(R)$
  $p = \lfloor \frac{R}{t_s} \rfloor;$
  IF $p \geq n$ THEN
    $p = n - 1;$
  ENDIF
  return(p);

The *linear* mapping function denoted by the subscript *lin* divides the range of R from 0 to nt_s evenly. t_s is a *tuning factor* whose value depends on the scheduling algorithm, so that the equation $max(R) = nt_s$ holds.

4.2.2 Scheduling Algorithms

For a real-time task T let,

$a(T)$ be the arrival time of T,

$e(T)$ be the estimated execution time of T,

$s(T)$ be the slack time given for T to complete, and

$d(T)$ be the deadline before which task T must complete.

Clearly, $d(T) = a(T) + e(T) + s(T)$ holds for any given task T. Thus, only three of these parameters need be known. For all tasks, the arrival time and the departure time are known, and the execution time estimates are required only by those scheduling algorithms based on the tasks' slack time.

The Earliest Deadline (ED) algorithm is emulated in two ways, the Earliest Deadline Relative (EDREL) and the Earliest Deadline Absolute (EDABS), while the Least Slack (LS) algorithm is emulated as Least Slack Relative (LSREL).

In EDREL, the priority level for a task is based on the deadline of the task *relative* to it's arrival time. Upon the arrival of a new task it's level is assigned by,

$$level = map_{lin}(d(T) - a(T))$$

Tasks with distant deadlines will naturally suffer, as they will be assigned lower priorities. When new jobs arrive with earlier deadlines, these tasks will not get the CPU even though their deadlines are fast approaching.

In EDABS, the algorithm assigns priority by computing the deadline of a task with respect to some fixed time t_{pinned}. This value is first assigned as the startup time and is later adjusted as per the system load conditions. When the number of tasks placed in the highest level $(n-1)$, i.e. the lowest priority, exceeds a *re-shift_count* N_r, the value of t_{pinned} is reset to the current time t. Such an occurance is called a re-shift. To avoid re-shifts due to a single task with unusually distant deadline, we only count the number of *consecutive assignments* of tasks to the $n-1$th priority level. The flag *pinned* is initialised

to *false* and upon arrival of a new task the following algorithm is employed:

IF ($not\ pinned$) THEN

  $pinned = true;$

  $t_{pinned} = t;$

  $max_assigns = 0;$

ENDIF

$level = map_{lin}(d(T) - t_{pinned});$

IF $level = n - 1$ THEN

  $max_assigns = max_assigns + 1;$

ELSE

  $max_assigns = 0;$

ENDIF

IF $max_assigns = N_r$ THEN

  $t_{pinned} = t;$

  $level = map_{lin}(d(T) - t_{pinned});$

  $max_assigns = 0;$

ENDIF

Upon task completion, t_{pinned} is set to *false* again if there are no tasks awaiting execution. Upon a re-shift, the new tasks are assigned priorities based on the more recent t_{pinned}, enabling them to obtain higher priorities. Thus, when a re-shift occurs, the older tasks will lose their advantage of having arrived early and are most likely to miss their deadlines. This is not unacceptable, as intuitively, when the re-shift occurs, the system is likely to be overloaded. Thus, giving up on some old tasks and starting afresh will help lessen the total number of missed deadlines. The rate of re-shifts is dependent on two factors - the load ρ present in the system, and the re-shift count, N_r. A high number of re-shifts will only result in poor performance, as more tasks are likely to be abandoned and miss their deadlines. But, if the number of re-shifts is too low, t_{pinned} would not change much and the system is likely to cling on to the older tasks, ensuring that neither the old

nor the new tasks meet their deadlines.

In LSREL, the priority of a task is calculated based on the task's slack at arrival. Thus, the priority level assigned is

$$level = map_{lin}(d(T) - a(T) - e(T))$$

4.3 Experimental Setup

The existing POSIX support on Linux has only 32 priority levels which could be used to assign priority to processes. We expanded this to accommodate 128 levels, the actual number of priority levels as per POSIX standards. To speed up the selection of a highest priority process to be scheduled, we indexed the whole set of POSIX processes using a 128 bit array. The *runq* is an array of 128 pointers to processes, which has all the POSIX scheduled processes in it. Further, we changed the underlying process scheduler from the default Linux scheduler to the Bach scheduler.

4.3.1 Linux Signal Handling Bug

We detected a bug in the signal handling portion of the Linux source code, as of release 1.2.3. In the distributed source code, whenever the scheduler is called, the real value of the time (in ticks) from the present, at which the next SIGALRM is to be generated is made equal to a very large value. After that, a check is done on all the processes to see whether a timer is used, and if so, when the signal scheduled is to be generated. The closest value is then assigned to a global variable *itimer-next*. To minimize the time spent in checking the timer values, a check is made to see if any time has elapsed since the last time the scheduler was called; if not, that particular process is not checked.

As a result, whenever there is more than one call to the scheduler within a tick, the timer value at which the next SIGALRM is to be generated was set to a very large value. Since the count of elapsed ticks is still zero, none of the processes are checked for their timers, and hence, the value of *itimer-next* remains a very large value. This bug resulted in a large loss of time due to signals being generated in our real-time experiments. However,

when the next call to the scheduler is done after some elapsed ticks, due to the CPU quantum for the current process being over, the *itimer-next* was set to the appropriate value. This situation may or may not occur in normal conditions, but is quite possible under real-time situations, as each process might not last more than a few millisecs. We fixed this bug before conducting the experiments reported later in this chapter.

4.3.2 System Calls Added

A few system calls were also introduced in order to facilitate our experiments. Some of these are related to the measurement of time in our experiments. The timing mechanism in Linux is based on the timer chip of the i486 mother board, which ticks at a constant rate. The counter in this chip is continuously decremented from a predetermined value until it reaches zero, and then the predetermined value is re-loaded and decremented again. This value can be read by latching the current value and then reading it into a register. We refer to this as the *latch count*, whose value can be anything between 11932 and 0. The *jiffies* is a global variable which is incremented once every 10 millisecs by the timer routine in the system. We implemented a new system call *GetTime* which returns the current time in the system in terms of *jiffies* and *latch count.* This provides the maximum accuracy to measure the elapsed time of a real-time task.

A new system call for *exit* was also introduced, which we refer to as *NewExit.* It does everything that the default *exit* does, but the child process does not wait for the parent process to acknowledge it's exit. Also, it does not return the completion status of the child process to the parent. *NewExit* does notify the parent of the child's completion before freeing up the process table entry. Note that the releasing of a process table entry is usually done by the parent, and until it happens the process state is set to TASK-ZOMBIE. This system call was necessary since our experiments involve forking off a large number of processes in quick succession. With the parent process tied up due to the forks to be done, the process table was not being vacated as fast as the new processes were created.

Since a process can not release itself, in *NewExit* we first do everything that the *exit* call does, assign the process to a variable, and then set a flag for releasing this process

before the next call to the scheduler. Whenever control returns from a system call or the *timer* routine, this flag is then checked and the *NewRelease* function called to release this process. The *NewRelease* function is similar in functionality to the normal *Release* system call, in that it frees up the proc structure passed as an argument to it. Before that, it does what a parent would have done if it had received the SIGCHLD signal, removes the links associated with the child process, and wakes up any process that is waiting for the child's exit.

4.3.3 Workload

We opted to use an experimental workload similar to that used by Adelberg *et al* in their simulation studies [1], [2]. They model real-time task arrival as a Poisson process, with arrival rate λ, which means that the inter-arrival times between tasks is distributed exponentially with mean value of $1/\lambda$. Task execution times are assumed to be normally distributed with mean μ and standard deviation σ. The slack values assigned for the tasks are uniformly distributed in $[S_{min}, S_{max}]$. We studied the performance of various scheduling algorithms under different system load conditions. Load, ρ, is defined as $\rho = \mu\lambda$ $(0 \leq \rho \leq 1)$.

Parameter	Simulation value	Experimental value
μ	0.5	100 millisecs
σ	0.1	20.0 millisecs
N_r	2.0	2.0
$[S_{min}, S_{max}]$	[0.1, 1.0]	$[20, 200\ millisecs]$

Table 4.1: Baseline parameter values

In all our experiments, the parameter values shown in Table 4.1 were maintained. The accuracy of the system clock was 10 millisecs; thus, for meaningful experiments, the mean execution times of the real-time tasks has to be in the order of several millisecs. We chose normally distributed task execution times with a mean of 100 millisecs. With respect to this value, the rest of the parameters were adjusted to reflect the correct load conditions.

The simulation study of Adelberg *et al* generated each data point by a run lasting either 100,000 time units or until a 95% confidence interval is obtained for the miss ratio to be within 1% of it's value [1], [2]. As our methodology involved measurements on real systems, we were not able to adopt such means, but instead ran our experiments until 10,000 tasks were assigned to the system and completed. We thus had a main process devoted to forking off tasks as children processes at predetermined periodic intervals. This main process was given the highest priority possible in the experimental setup, while the children processes got their priorities based on their nature and the scheduling algorithm used. Each child process ran a single program which took a few arguments including the time for which it is supposed to use the CPU. It had a 100 microsecond and a 10 microsecond computational loop to allow child processes to have CPU usage requirements to a granularity of 10 microseconds.

As the parent has no control over the child process after the forking off, the child must determine whether it can complete its task before the deadline and report back to the parent. Also, for some of our performance measures, if the child misses a deadline, it must notify its parent of the time by which it missed the deadline and the slack value assigned to it. This communication could not be done through file I/O, as we determined by experimentation that the file control operations took a few milliseconds to complete. Instead, we used message passing by the child process using a global message queue. The overhead involved was on the order of tens of microseconds.

4.3.4 Fine Tuning

For strict time control, time should start ticking for a child process as soon as it is forked off, i.e., the parent should record the start time of each child process. If instead the child were to record its own start time when it starts executing, there is a distinct possibility of a large disparity between the actual process creation time and the recorded creation time. Since there is only one parent process that forks off all the real-time children, it is necessary that this parent process has the highest priority, so that it can get control over the CPU as and when it requires to fork off a child. To avoid any unnecessary work on

the part of the parent during the course of the experiment, before forking off, we compute the execution time and slack time for all the 10000 children along with the inter-arrival time with respect to the previous child and store it in a data structure before the start of the experiment.

Further, on a fork, the elapsed time between the previous fork and the current time is noted and compared with the pre-determined inter-arrival time. If the elapsed time is larger, the next inter-arrival time is reduced by the amount of difference in order to maintain the accuracy of the workload generated. If this compensation is not done, we would start accumulating all the late arrivals and allow the system to be actually less loaded than the intended load.

Certain other Linux source code corrections were also required. The original *Setitimer* system call actually sets the timer to send SIGALRM after the elapse of the given number of ticks *plus one*. Also, the code which checks for the expiry of a timer looks for the elapsed ticks to be *greater* than timer ticks, rather than for equality

Despite these attempts, the parent continued to get control of the CPU approximately a tick away from the intended time, which could not be avoided. This is due to the granularity of the system clock (10 millisecs); if you want to fork off a process 73 millisecs from now, the timer is set to 80 millisecs and the parent gets control only *after* 80 millisecs.

While working with the child processes, we found that for every fork there was a considerable amount of time spent by the operating system in setting up the child before giving control to it. When the relevant portions of Linux source code was studied, we found that the system call to open the library *libso.4* took several milliseconds. This was avoided by having the child program compiled as static, so that the library is linked along with the executable during compile time rather than dynamic linking during execution time. This also helped in reducing the number of page faults incurred.

4.3.5 Performance Measures

The performance measure used in the literature is the *miss ratio*, [1], [2], defined as the ratio of the number of tasks that miss their deadline to the total number of real-time tasks

that were initiated. In addition to using miss ratio for evaluating the performance of real-time scheduling, we used two other parameters.

Slack Range	Weightage	Slack Range	Weightage
0.10 - 0.19	1.0	0.55 - 0.64	0.5
0.19 - 0.28	0.9	0.64 - 0.73	0.4
0.28 - 0.37	0.8	0.73 - 0.82	0.3
0.37 - 0.46	0.7	0.82 - 0.91	0.2
0.46 - 0.55	0.6	0.91 - 1.00	0.1

Table 4.2: The weightages assigned for different slack values

The first was a *weighted miss ratio* in which those tasks' that missed deadlines were given weightage according to their importance. We infer the importance of a task from its slack value. The slack time range for all the tasks was divided evenly into ten parts. A low slack value means that the task must be completed urgently. Accordingly, when computing the weighted miss ratio, a task whose slack value falls in the lowest slack range is weighted to contribute *fully* to the total number of misses, while a task which falls in the highest slack range is weighted to contribute less to the total number of misses. Table 4.2 shows the weightage values for the different slack ranges.

As with miss ratio, the lower the value of the weighted miss ratio, the better the performance of a scheduling algorithm. In our experiments slack values were assigned uniformly distributed between 0.1 and 1.0. Ideally, we would like to see the missed tasks falling uniformly in the slack range, i.e., if x tasks missed their deadlines with the slack value in the lowest range, the same number of misses should be repeated across the whole slack range. Thus in the ideal case the *weighted miss ratio* will have a value of 0.55 *times miss ratio*. But, this may not be the case due to our assumption of exponential inter-arrival times, which results in the system load occasionally being very heavy. Also, as a lower value of slack time means that the task is of greater importance, the real-time system is expected not to miss any of them. Thus, we expect the weighted miss ratio to be less than 0.55 *times miss ratio*. For all tasks that have missed their deadlines, the slack value is noted and the weighted miss ratio is calculated from

$$\left(\sum_{i=1}^{n} weightage(slack_value_i) \right) / (total\ number\ of\ tasks)$$

The weightage function could in general be any kind of function, not necessarily linear. This measure tells us if we are missing an unusually large number of tasks which have low slack values.

The second performance measure that we used was *relative miss ratio*. This gives an idea of the difference in time between the *deadline* and the *actual* time at which the task completed. Since the tasks' have soft deadlines, even if the system misses the deadline for a particular task we would like it to continue execution and complete after some time. So it is important for us to know how a scheduling algorithm behaves in order that tasks which miss their deadlines do not finish very late. It should not be the case that the miss ratios for a particular scheduling algorithm are low, but its *relative miss ratio* is high, which means that the low miss ratio was obtained by sacrificing tasks that have missed their deadlines.

We compute the relative miss ratio for all tasks that missed their deadlines as

$$\sum_{i=1}^{n} \left(\frac{time\ by\ which\ the\ deadline\ was\ missed}{(slack\ time + execution\ time\ )} \right) / (total\ number\ of\ tasks)$$

4.4 Results

Figure 4.1 shows experimental results for the POSIX system with 128 priority levels. The intra-level scheduling algorithm used in this set of experiments is First In First Out (FIFO), with the tuning factor t_s set to the optimum value for EDABS, which was 0.03. For EDREL and LSREL, the t_s value was set according to the equation $max(R) = nt_s$, where n is the number of priority levels. The figure shows three graphs for the three evaluation parameters, viz., *miss ratio*, *weighted miss ratio* and *relative miss ratio*. Similarly, the right hand side shows the graphs for high system load conditions. We see that in terms of miss ratio all three algorithms perform more or less equally, while for the other two

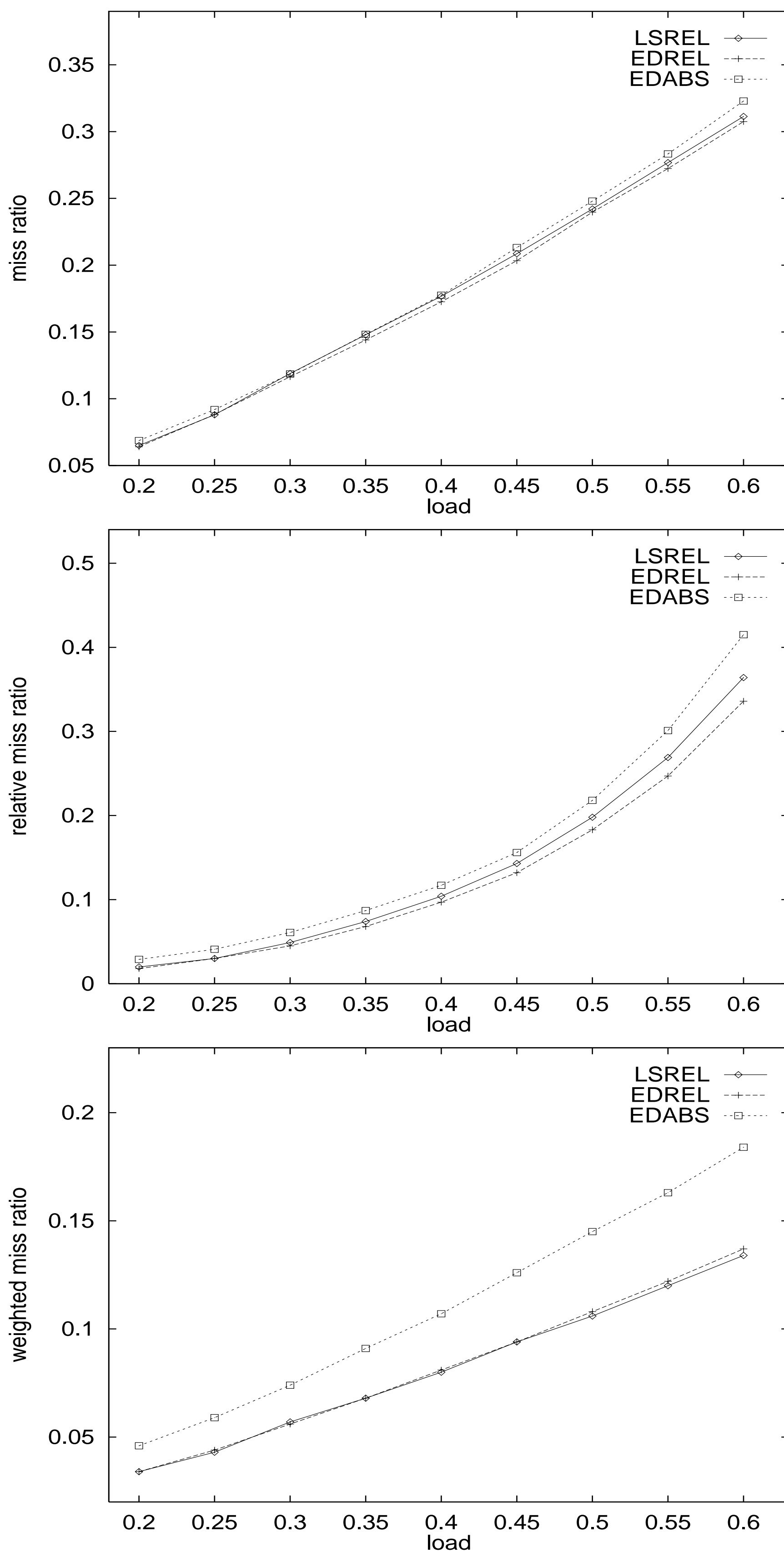


Figure 4.1: POSIX system with 128 priority levels: real-time workload.

parameters EDABS performs the worst. For weighted miss ratio in particular, we see a significant difference in values. This means that EDABS is achieving it's otherwise comparable performance at the cost of tasks which have low slack value. The reason for such poor performance is that the arbitrary abandoning of the tasks, without any concern regarding their importance in terms of their low slack values, leads to greater weighted miss ratios.

Figure 4.2 shows the performance of the emulated algorithms on a traditional Unix system with only 40 levels of priority. The difference in terms of the lower number of priority levels and the re-assigning of task priorities by the OS scheduling algorithm, has negligible effect on performance. The mean execution time for a task is 100 millisecs and the quantum assigned to a task by the scheduling policy, Multi-Level Round Robin with Feedback, is also 100 millisecs. Also, the decaying of the CPU usage of a process is done once every second. Most processes complete even before their priority can be changed or they are context switched out. Thus, the OS scheduling does not affect the performance of the emulated real-time scheduling algorithms.

Next we look into the effects of Round Robin intra-level scheduling in a POSIX system through Figure 4.3. Here, the scheduler slice (Schslice) quantum assigned by the scheduler is varied with a fixed mean execution time for the tasks. The experiments were carried out under a load of $\rho = 0.65$. The graphs are plotted for the three performance measures; we notice that as the Schslice/mean ratio increases, the performance of the algorithms tend to improve. This is because the greater the quantum assigned to a task, the greater is the probability of a task completing before being context switched out of the CPU. Thus, it is always advisable to assign a large quantum in the case of a Round Robin intra-level scheduler.

Consider that there is an optimal value of the tuning factor for which the number of misses is the minimum for a given load value, all other factors remaining the same. Under EDABS, when the tuning factor value is lower than this optimal value, the number of re-shifts is higher, resulting in higher miss ratios. This is because the priority values (p in the map_{lin} function) now being calculated will be numerically higher, resulting in the

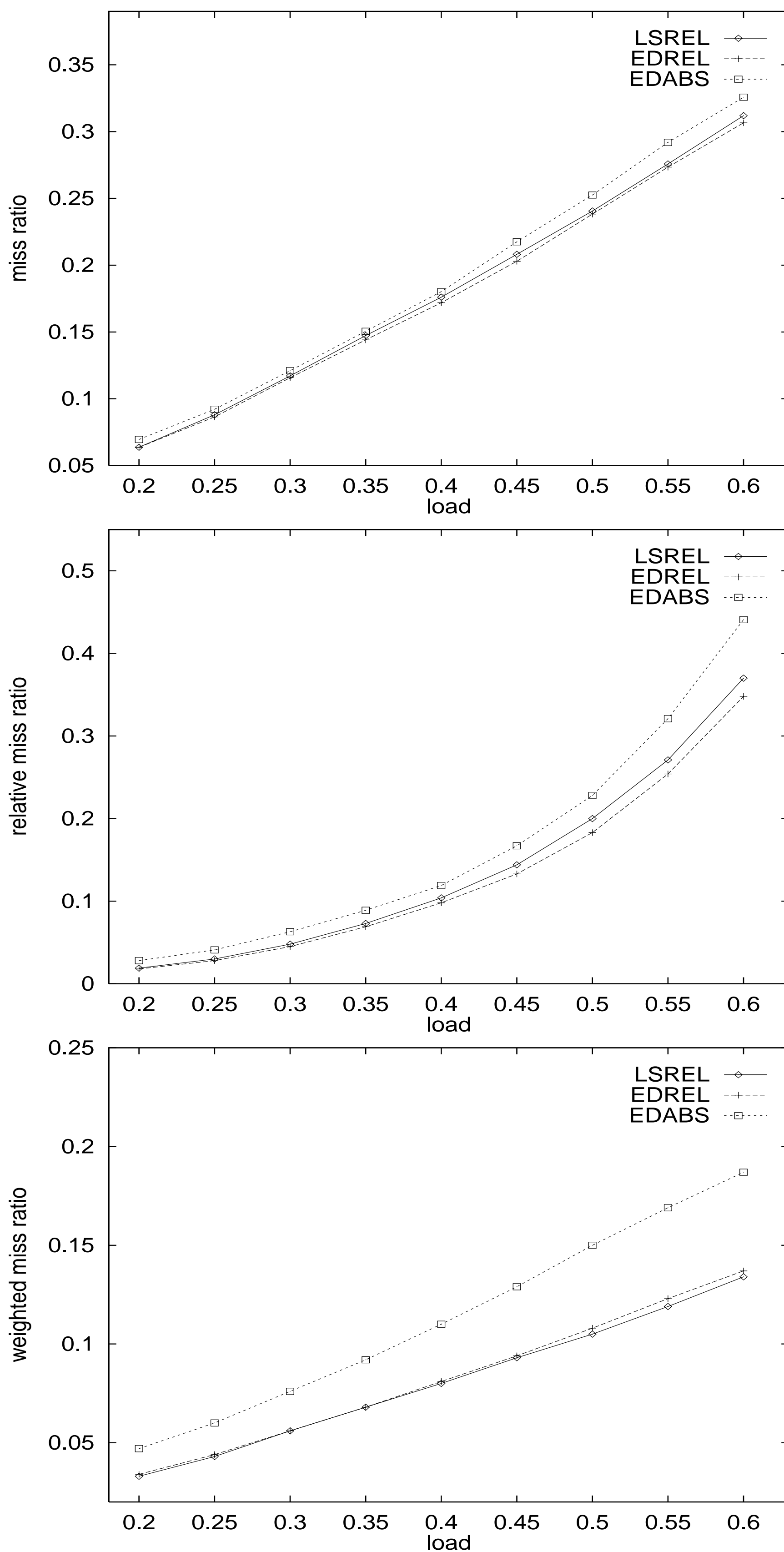


Figure 4.2: Traditional Unix system with 40 priority levels: real-time workload.

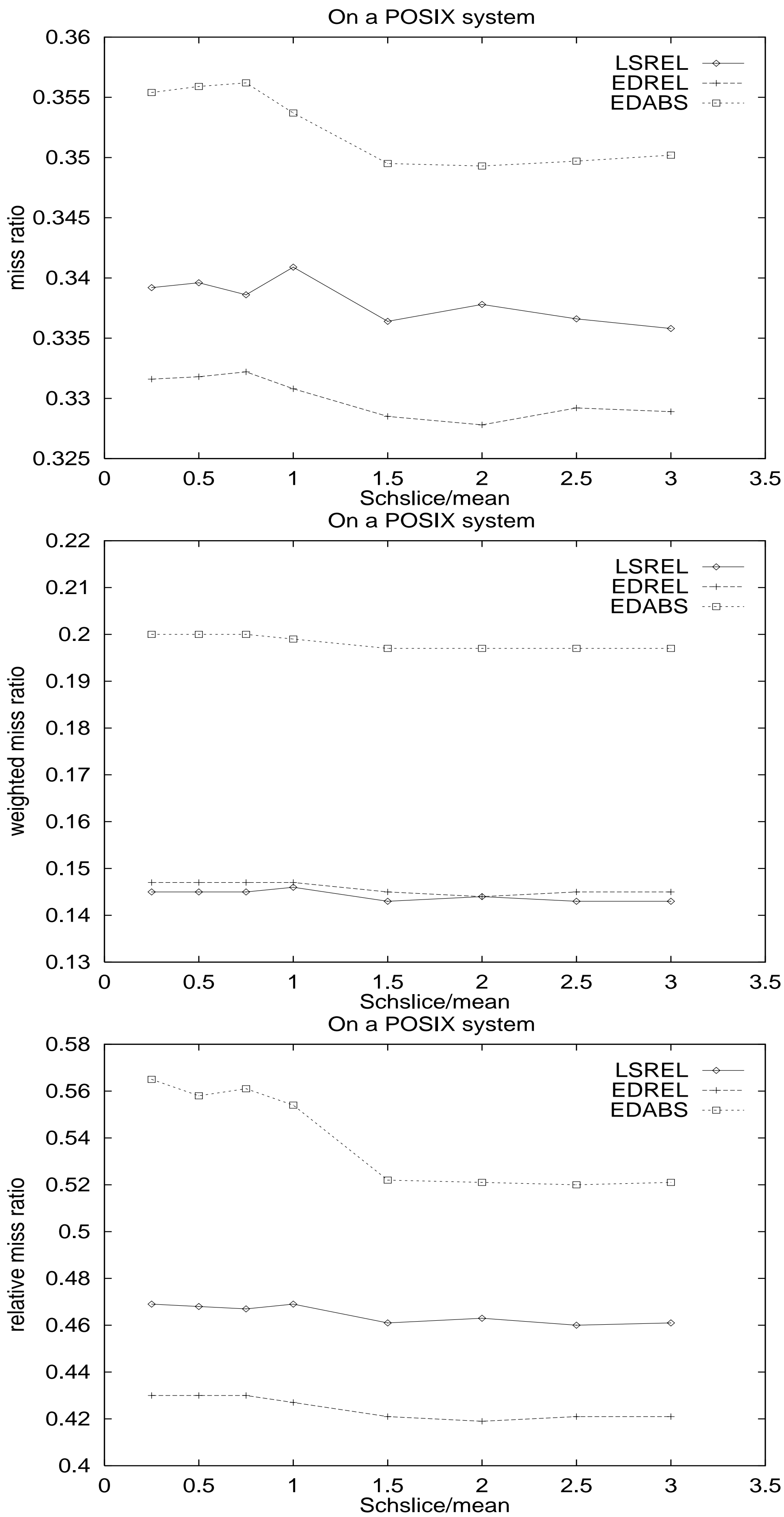


Figure 4.3: Round Robin scheduling in a POSIX system: real-time workload

highest priority level being reached more often. Note that by highest priority level we mean that the lowest priority value is assigned to the task, whatever might be the system under consideration. Therefore, a re-shift occurs which actually results in the re-allocation of priority values, with the older processes losing their advantage of having arrived first. On the other hand, if the tuning factor (t_s) value is higher than optimal, then as the load increases there may not be sufficient re-shifts for good miss ratios. Basically, the sacrificing of a few processes in order to get better miss ratios by doing a re-shift would not occur.

This holds for both EDREL and LSREL, although the tuning factor affects the number of misses in different ways. The value of t_s is got from the equation $max(R) = nt_s$; if the t_s value varies significantly from this value, then either the whole priority range will not be used by the system due to the large value, or the priority of a task will often go beyond the available range and hence the task will be assigned to the last available level. In either case, the performance of the system will deteriorate; in the first case, there is heavy competition for a limited subset of priority levels even though there are enough levels, and in the second case the last available level will most often be used, identically to the basic intra-level scheduling available in the system. Thus, it is important that the tuning value be optimised for the load conditions and work patterns of the system, for the performance of the system to remain near optimal.

Figures 4.4 and 4.5 show results from our experiments with varying the t_s value on a POSIX system with FIFO intra-level scheduling, under a system load of 0.30 and 0.65 respectively. It should be noted that the x-axis is a logscale in both these figures. For EDREL and LSREL the t_s values based on the equation $max(R) = nt_s$, are 0.014 and 0.0078 respectively, for 128 priority levels. These were the default t_s values used for EDREL and LSREL in all experiments conducted. As regards EDABS, the t_s value which gave the lowest miss ratio, 0.03 was used. In our experiment, the variation in t_s was around these values.

As for as LSREL and EDREL are concerned, the resulting curves are almost identical under both the load conditions. Within these graphs the weighted miss ratio more or less sticks to the same kind of curves as the miss ratio. The explanation for this 'bucket' shape

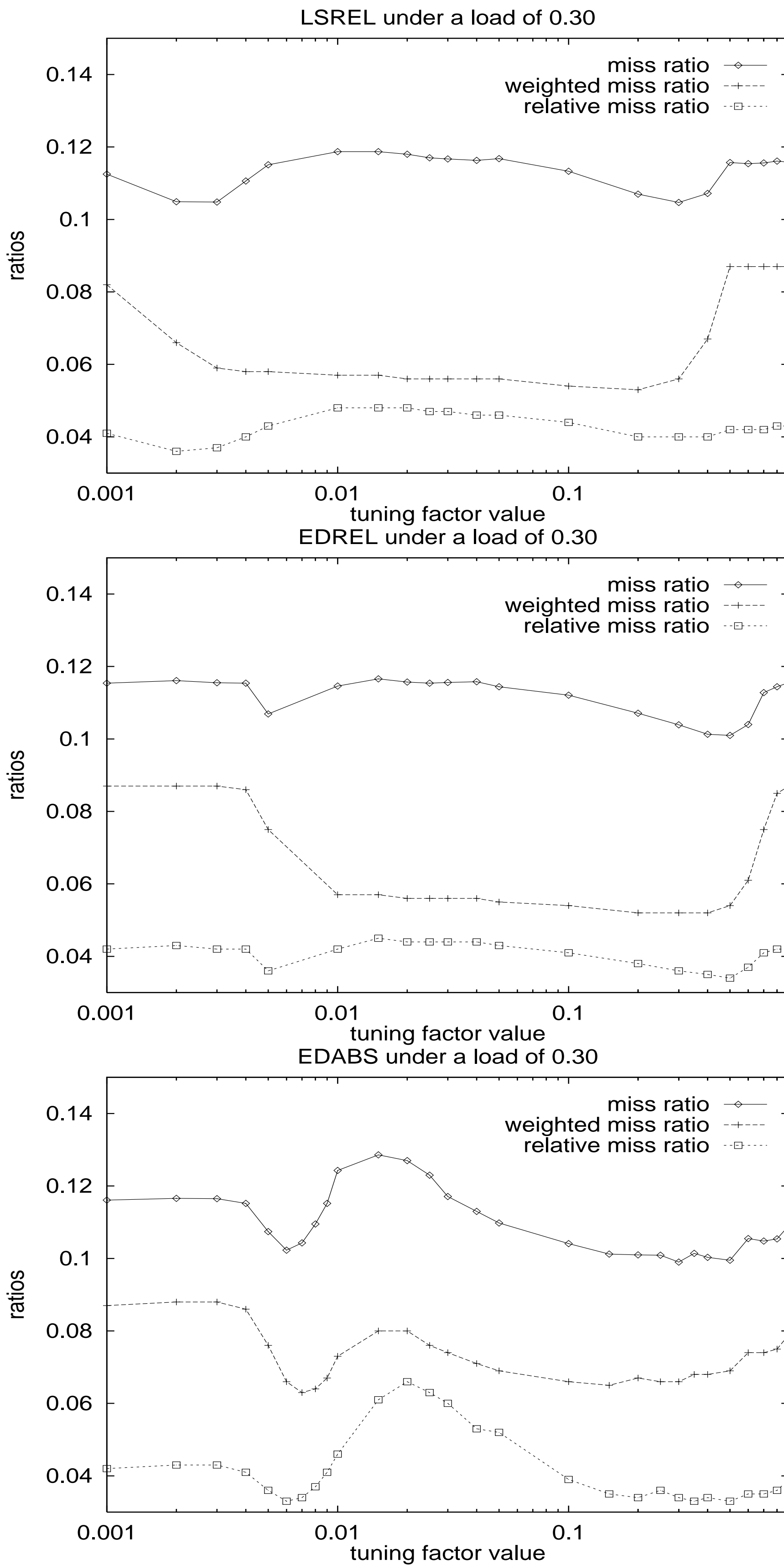


Figure 4.4: Tuning the algorithms in a POSIX system: load of 0.3

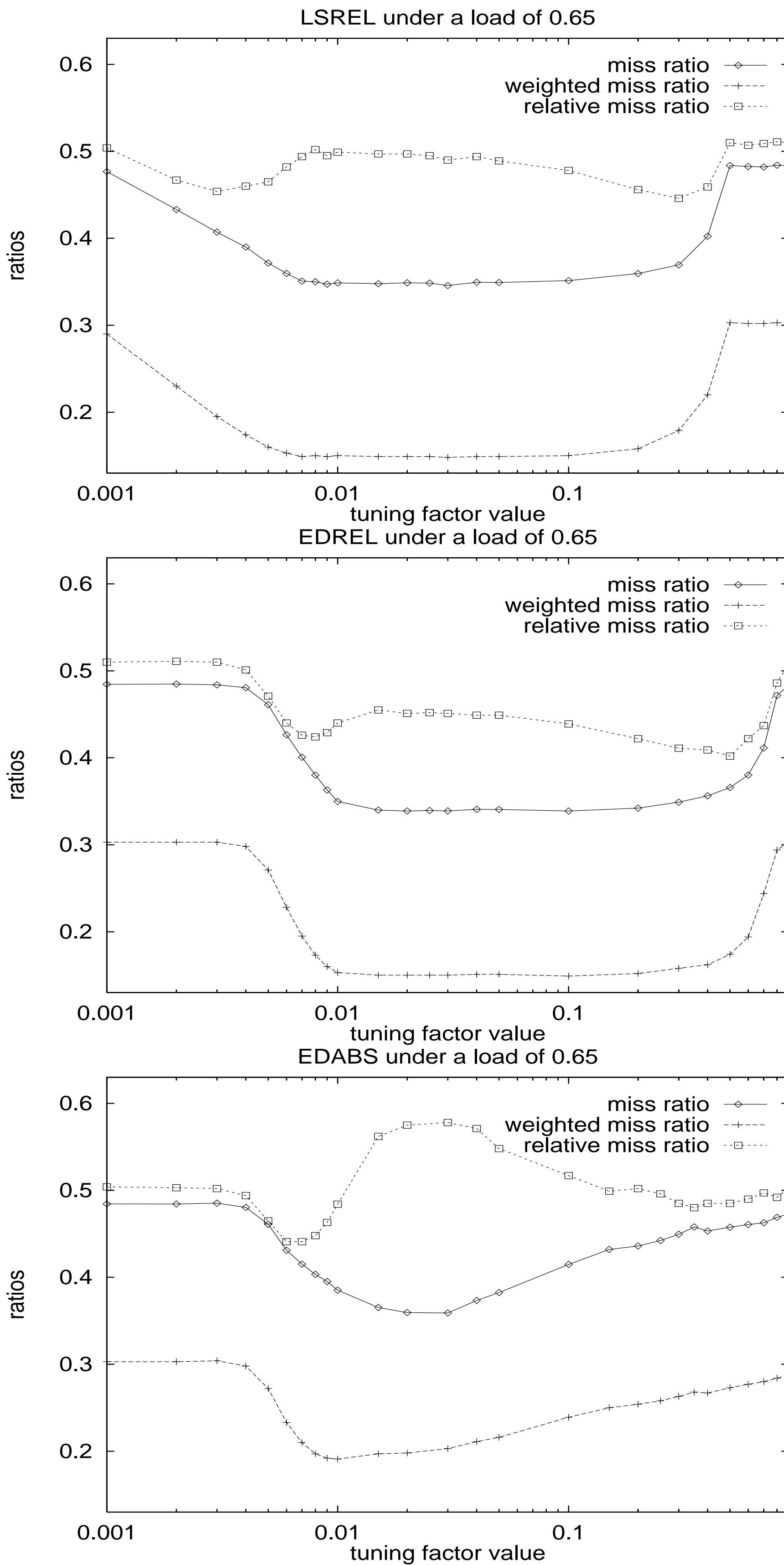


Figure 4.5: Tuning the algorithms in a POSIX system: load of 0.65

is as follows – recall that the map_{lin} function which we defined earlier, is used to map the real value of the slack or deadline of a task to a priority level. Thus, as t_s increases, the number of priority levels on to which the mapping happens steadily decrease and finally come down to just one or two priority levels. This means that the scheduling is now being applied to the tasks more or less with FIFO intra-level scheduling. Similarly, as t_s decreases. the mapping function is almost always going to exceed the available priority levels and will be forced to assign the last priority level available to the real-time task. This again means that all the tasks are assigned to the same priority level, resulting in FIFO scheduling. The relative miss ratio has it's best (lowest) values somewhere at the start of the steep increase in the miss ratio and weighted miss ratio curves. Note that all the measures reach a saturation level after the steep climb and that there is almost no change in the values thereafter. At these saturation points, in the case of 0.65 load, the miss ratios are quite high (around 0.5) resulting in almost every other process missing it's deadline.

With respect to EDABS, it is interesting to note that due to the presence of re-shifts, we see that the t_s value significantly affects the performance of the algorithm, indicating that the value must be carefully chosen. The best t_s value under low system load condition is not the best value under high system load conditions. Also, for a load of 0.65, the *relative miss ratio* and the *miss ratio* curves are the mirror images of each other for most part, implying that the t_s value has to be chosen depending on what performance measure we are more concerned about. In retrospect, this peculiarity was to be expected, as the EDABS algorithm gives up older tasks by performing re-shifts to lessen the number of missed deadlines.

We define the *re-shift ratio* as the ratio of the total number of re-shifts that occur to the total number of tasks that were initiated by the system. Figure 4.6 shows the re-shift ratios for the case of EDABS when the tuning factor was varied. The reason for the dip at the start of the x-axis in Figures 4.4 and 4.5 for EDABS is explained when we see this graph - the t_s value is so low that the system always performs a re-shift and assigns the new task to the last priority level. This amounts to the FIFO algorithm being employed

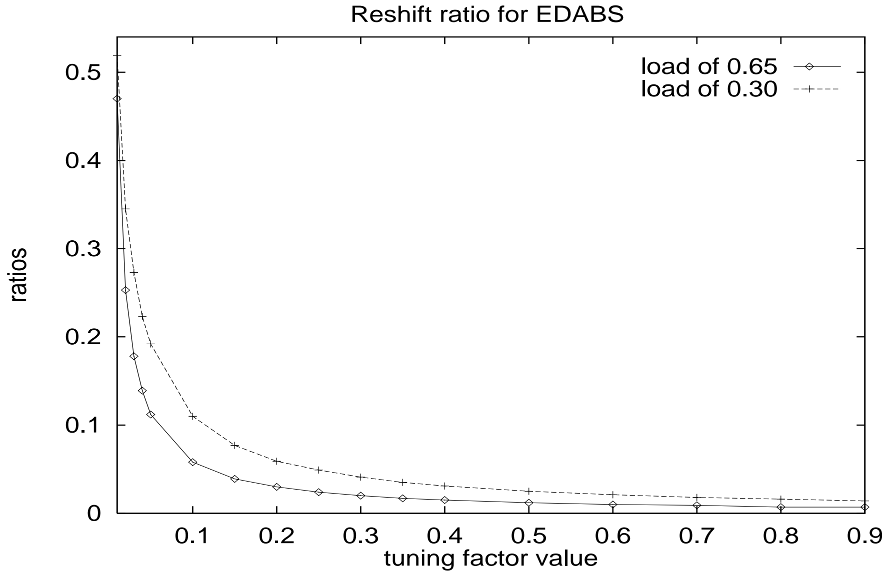


Figure 4.6: Re-shift ratios for EDABS on a POSIX system

on the tasks at the start of the x-axis, which is why we see the horizontal line, implying the ineffectiveness of the large reshifts that occur. Later the curves dip as the benefits of doing re-shifts ensure a lower miss ratio. After that, the climb in miss ratio values is immediate for most. This is because, the effects of EDABS and the small number of priority levels available, combine to peak the graph for all the measures except the *miss ratio* and *weighted miss ratio* in the case of 0.65 system load. As the tuning factor value further increases we see the beneficial effects of less re-shifts in terms of the miss ratios decreasing. Thus as the *re-shift ratio* goes down, with exception of the miss ratio and weighted miss ratio under 0.65 system load condition, all others attain favourable values (numerically lower).

In short, EDABS performs poorly in most of our experiments. To confirm this observation, we conducted a simulation study with the same workload, under the same parameter values and found that it should actually perform better than EDREL, as reported by Adelberg *et al* [1], [2]. Figures 4.7 and 4.8 show the simulation results under different system loads. We see that EDABS tracks Earliest Deadline first (ED) very closely and out-performs it under high system loads. Also, in most cases EDABS is the best of the emulated algorithms, except in the case of *weighted miss ratio* where it is the worst. As we explained earlier based on our experiments, this is due to the arbitrary abandoning of

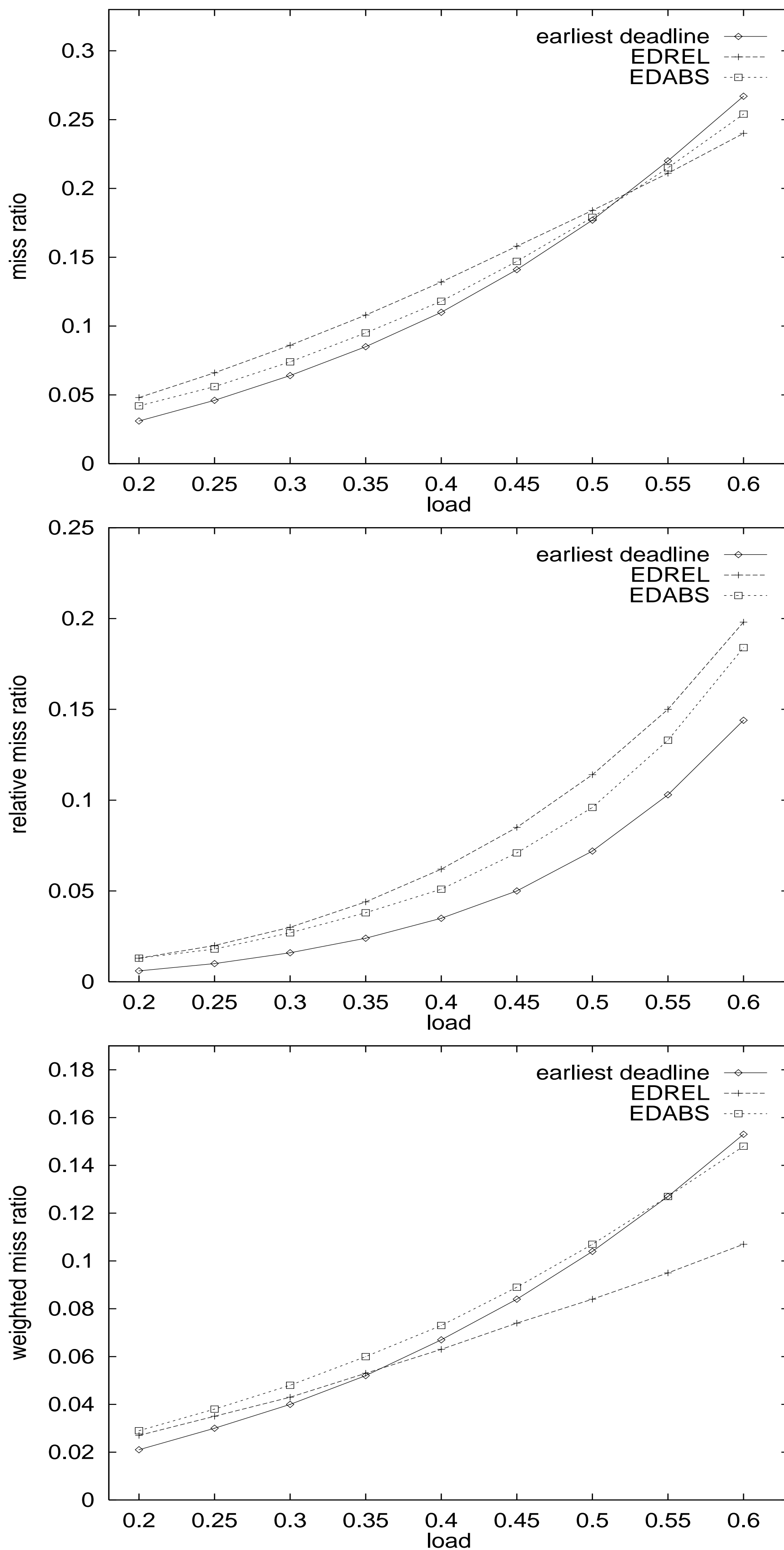


Figure 4.7: Simulation of earliest deadline algorithms on a POSIX system

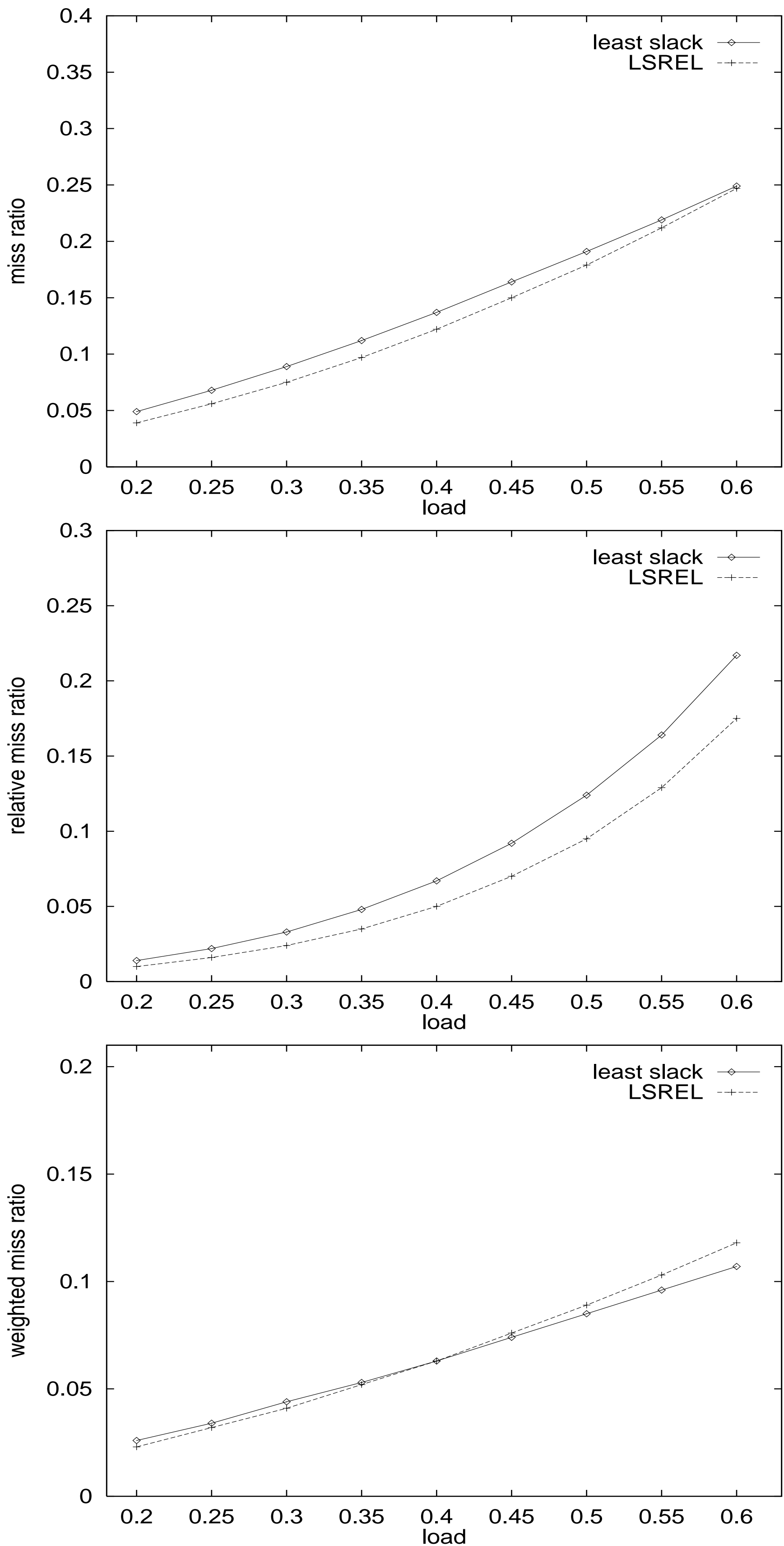


Figure 4.8: Simulation of least slack algorithms on a POSIX system

tasks when a re-shift occurs. Also, comparing Figures 4.1 with Figures 4.7, 4.8, we see that there is a significant amount of difference between the experimental and simulation results. This is mainly due to the overheads involved in the experiments and due to the system clock granularity; the simulation assumed zero overhead costs.

Algorithm	**Experimental value**			**Simulation value**		
	miss ratio	*weighted miss ratio*	*relative miss ratio*	*miss ratio*	*weighted miss ratio*	*relative miss ratio*
EDABS	0.284	0.161	0.366	0.275	0.155	0.313
EDREL	0.280	0.122	0.295	0.269	0.118	0.261
LSREL	0.289	0.122	0.324	0.280	0.119	0.286

Table 4.3: Miss ratios for a POSIX system under experimentation and simulation.

In order to confirm this explanation and the correctness of our experiments, we ran an experiment with task mean execution time set to 1 second. With such a large value of execution time, the clock granularity of 10 millisecs plays a negligible role in affecting the performance of the system. Table 4.3 shows the simulation values and the experimental values of miss ratios for all the scheduling algorithms on a POSIX system, run for 10000 tasks. We see from the table that the difference between simulation and experimental values for the scheduling algorithms is negligible – within 3% for miss ratio and weighted miss ratio, while it is up to 15% for relative miss ratio. It should be noted that the system load was set to 0.65 and the mean execution time of a task was 1 second (100 times the clock accuracy of 10 millisecs). These, along with the experimental overheads, might be the reasons for such large discrepancy. If the mean execution time of task is further increased, the overall system overhead per task will also increase, but the negative effects of clock granularity will decrease. However, we are not concerned with real-time tasks with mean execution times of 1 second and beyond. This experiment was conducted merely to demonstrate the correctness of the other experiments.

We thus believe that the reason for the disparity between experimentation and simulation results is partly related to the system clock granularity of 10 millisecs. On average, the processes actually arrive 5 millisecs later than their intended arrival time, mainly

because the forking of the processes is done by a single process, which wakes up every now and then. Recall that as we wanted the experiments to closely adhere to the simulated workloads, we computed the actual inter-arrival time between processes and then compensated for the late arrival, if any, by subtracting the amount of time from the next inter-arrival time. Thus, we have a situation where the process arrives at a time quite different from the intended arrival time, resulting in the relative difference between t_{pinned} and the current system time to be off the mark. This means the priority is quite different from the otherwise simulated value. A re-shift might occur because of this, resulting in some more complexity. These deviations are quite disastrous in the case of low tuning factor values, since even small time differences result in significant change in the priority values. It should be noted that even though we do the same kind of compensation for LSREL and EDREL, they are not affected that much by these varying arrival times. This is because the slack time upon which the priority is computed for LSREL never changes for a given task and the deadline (slack plus execution time) which is relative to the arrival time also does not change.

4.5 Conclusion

In this chapter, we first defined two new measures for evaluating soft real-time scheduling algorithms - the *weighted miss ratio* and the *relative miss ratio*. We saw that the number of priority levels, at least between 40 and 128, does not play a significant role in the performance of the algorithms. Also, the operating system scheduling policy had no effect as long as the CPU quantum assigned to a process was large enough and the priority re-computation, if any, was delayed as far as possible. We also looked into the effects of Round Robin intra-level scheduling and came to similar conclusions. In our simulations studies of these algorithms, we observed significant deviation in the behaviour of the EDABS algorithm. It was the best of the simulated algorithms, but performed worst in our experimental studies. This we attribute to the coarseness of the system clock and the inability of our experimental setup to accurately fork off a task at the intended time.

Chapter 5

Conclusion

This thesis studied two aspects of Unix decay-usage schedulers - their fairness and their suitability in soft real-time environments. To quantify the fairness of Unix process schedulers experimentally, the Linux operating system's process scheduling and related code were modified. The modifications provide for the traditional decay usage process scheduling of an Unix system and 4.3 BSD like process scheduling, both built on top of a Linux system. To study soft real-time scheduling, the POSIX standard process scheduler was also implemented. To facilitate real-time experiments, support for fast release of system resources and for measuring elapsed times were provided using new system calls.

5.1 Results and Contribution

In the literature, there exists no clear-cut mechanism to evaluate the *fairness* of a process scheduling algorithm in *real-world* situations. In the first part of this work, we quantified fairness to evaluate various decay usage process schedulers. We conclude (i) that the 4.3 BSD scheduler appears to be superior to the other schedulers studied, and (ii) that the fairness of a Bach scheduler can be improved upon without any significant overhead, if the priority is recomputed on a context switch. We saw that although the SUN Sparc20's SVR4 scheduler was relatively unfair, it is admirably consistent in its treatment of processes. We found the Bach scheduler to be the most inconsistent scheduler of all. We realised that

this observation may have been due to the fact that the start of a decay period was not always aligned with the start time of the experiment, and the mis-alignment varied across experiments.

In the second part of our work, we studied soft real-time process scheduling. We defined two new measures for evaluating the emulated scheduling algorithms. We were able to infer the tradeoffs involved from these two measures, as they clearly portrayed where exactly the performance gain or loss of an algorithm comes from. Although EDABS performed best in our simulation studies, it proved not to be the best scheduling algorithm in our experimental studies. We believe that this was due to the inability of our experimental framework to provide a sufficiently accurate clock.

5.2 Further Work

Further work in this area could be done in the area of the fairness of the SUN Sparc20's process scheduler. This can be done by first analysing the relation between the various parameters in its scheduling table (Table 2.2 in Chapter 2) and the usual decay usage scheduler parameters. Improvement is clearly possible, since the user has control over all the values of the parameters in the table.

Further, in soft real-time systems, we saw that the EDABS scheduling algorithm is highly dependent on system load and the tuning factor value, t_s. This suggests the development of algorithms which dynamically change the tuning factor value depending on the system load conditions. This would initially require detailed simulation of the system under various workload patterns, followed by a study of the relation between the best value for t_s and the given load condition. To do this work on our experimental setup would require a more accurate clock - a complete re-design of the signal handler and timer mechanism could be attempted to provide accuracies down to a few microseconds.

Bibliography

[1] B. Adelberg, H. Garcia-Molina, and B. Kao. Emulating Soft Real-Time Scheduling Using Traditional Operating System Schedulers. Stanford University Technical Report, 1994. Available by anonymous ftp from db.stanford.edu in /pub/adelberg/1994/ as priority2.ps.

[2] B. Adelberg, H. Garcia-Molina, and B. Kao. Emulating Soft Real-Time Scheduling Using Traditional Operating System Schedulers. In *IEEE Real-Time Systems Symposium*, pages 292–298, December 1994.

[3] M. J. Bach. *The Design of the Unix Operating System*. Prentice Hall India Ltd., 1989.

[4] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck. Supercomputer Performance Evaluation and the Perfect Benchmarks. In *Proceedings of the International Conference on Supercomputing*, pages 254–266, 1990.

[5] D. H. J. Epema. An Analysis of Decay Usage Scheduling in Multiprocessors. In *Proceedings of the ACM SIGMETRICS Conference in Measurement and Modeling of Computer Systems*, pages 74–85, May 1995.

[6] R. B. Essick. An Event-based Fair Share Scheduler. *Proceedings of USENIX*, pages 147–161, Winter 1990.

[7] B. Furht, J. Parker, D. Grostick, H. Ohel, T. Kapish, T. Zuccarelli, and O. Perdomo. Performance of REAL/IX - Fully Preemptive Real Time UNIX. *Operating Systems Review*, 23(4):45–52, 1989.

[8] D. Gillies. comp.realtime: Frequently asked questions (faq). http://www.cis.ohio-state.edu/hypertext/faq/usenet/realtime-computing/faq/faq.html, May 1996.

[9] B. M. Goodheart and J. H. Cox. *The Magic Garden Explained: The Internals of UNIX System V Release 4*. Prentice Hall, Eaglewood Cliffs, NJ, 1994.

[10] J. L. Hellerstein. Achieving Service Rate Objectives with Decay Usage Scheduling. *IEEE Transactions on Software Engineering*, 19(8):813–825, August 1993.

[11] G. J. Henry. The Fair Share Scheduler. *AT&T Bell Laboratories Technical Journal*, 63(8):1845–1857, October 1984.

[12] M. K. Johnson. The Linux Kernel Hackers' Guide. Document available for ftp from any linux site, under the directory /pub/linux/docs/linux-doc-project/kernel-hackers-guide/ as khg-0.6.ps.gz, April 1995.

[13] J. Kay and P. Lauder. A Fair Share Scheduler. *Communications of the ACM*, 31(1):44–55, January 1988.

[14] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, MA, 1988.

[15] Y. Mizuhashi and M. Teramoto. Real time operating system: RX-UX 832. *Microprocessing and Microprogramming*, 27:533–538, 1989.

[16] S. Shet. Unix Load Monitoring. M.E. Report, January 1994.

[17] G. A. Wainer. Implementing Real-Time services in MINIX. *Operating Systems Review*, 29(3):75–84, July 1995.

[18] G. Wells. A comparision of four microcomputer operating systems. *Real-Time Systems*, 5:345–368, 1993.