Incorporating fair share scheduling into the Layered Queueing Network Model

By

Lianhua Li

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science in Information and Systems Science

Department of Systems and Computer Engineering

Carleton University

Ottawa, ON, Canada, K1S 5B6

January 15, 2009

Your file Votre référence
ISBN: 978-0-494-52019-2
Our file Notre référence
ISBN: 978-0-494-52019-2

Abstract

Fair share scheduling is widely used in many applications. The impact of a fair share scheduler to the performance of an application is of concern to many researchers. To study the effects of fair share scheduling, a scheduler supporting share caps and guarantees and task groups has been inserted into PARASOL, a discrete event simulation engine. This scheduler is modeled on the Completely Fair Scheduler (CFS) found in Linux.

The Layered Queueing Network (LQN) performance model is an important method for studying software performance issues. Extensions are made to the LQN model to support fair share scheduling. The LQN simulator, Lqsim, built upon PARASOL, is extended to support these extensions.

This thesis provides useful explorations of performance impacts of CFS scheduling to an experimental system. The guarantee and cap share act differently in the application in that guarantees may be affected by the interaction between tasks while caps are not.

Acknowledgements

I would like to express my great thanks to my supervisor Dr. Gregory Franks for his help and support. He gave so much valuable guidance in the process of my research.

Meanwhile, many people gave me useful advice in my research; I like to express my thanks to them. I am also very appreciative of the help offered by the staff of the SCE department.

Lastly, I would like to express my special thanks to my husband Park and my dear son Tom. They provided me the endless love, encouragement and support throughout the entire process.

Table of Contents

Abstract.....i
Acknowledgements.....ii
Table of Contents.....iii
List of Tables.....vii
List of Figures.....viii
List of Listings.....x
Glossary of Terms.....xi
Chapter 1 Introduction.....1
1.1 Motivation.....2
1.2 Contributions.....3
1.3 Thesis Organization.....4
Chapter 2 Background.....5
2.1 Fair share scheduling.....5
2.1.1 Share.....6
2.1.2 Guarantee and Cap.....7
2.1.3 Types of Fair share scheduling.....8
2.1.3.1 Fair queuing (FQ).....8
2.1.3.2 Weighted Fair Queuing (WFQ).....8
2.1.3.3 Proportional-share scheduling.....9
2.1.4 Related works of fair share scheduling.....10
2.2 Completely fair scheduling.....12
2.2.1 CFS group scheduling.....13
2.2.2 The Linux CFS scheduler.....14
2.2.3 Red-Black Trees.....15
2.3 Layered Queuing Network (LQN).....16
2.3.1 Elements in LQN model.....17
2.3.1.1 Processors.....17
2.3.1.2 Tasks.....18
2.3.1.3 Entries.....19

2.3.1.4 Activities.....20
2.3.1.5 Precedence.....20
2.3.1.6 Requests.....20
2.3.2 Graphical Notation of LQN 21
2.3.3 XML Schema of LQN Component.....22
2.3.4 Old Style LQN Input Model.....25
2.4 LQN simulation tools.....26
2.4.1 Components of a Discrete Event driven Simulation.....26
2.4.2 Parasol Simulation Engine.....28
2.4.2.1 PARASOL Internels.....31
2.4.2.2 PARASOL built-in scheduler.....35
2.4.3 Lqsim Simulator.....36
2.4.3.1 Reference task.....37
2.4.3.1 Server task.....37
2.5 Summary.....38
Chapter 3 CFS scheduling in LQN.....39
3.1 CFS scheduling algorithm.....39
3.1.1 The Basic CFS scheduler.....39
3.1.2 The CFS group scheduler.....41
3.1.3 The Mixed CFS scheduler.....42
3.1.4 Normalize the shares.....43
3.2 Integration CFS into the LQN model.....43
3.2.1 Extension to LQN model.....44
3.2.2 Extension to LQN Core schema 46
3.2.3 Extension to Old Style LQN Input Model.....48
3.3 PARASOL CFS scheduler.....50
3.3.1 Parasol CFS ready-to-run-queue.....50
3.3.1.1 Host-rq and group-rq.....51
3.3.1.2 Structure of a CFS-rq and nodes in a CFS-rq.....53
3.3.1.3 Operations of a CFS-rq.....54
3.3.2 Parasol Built-in CFS scheduler.....58

3.3.2.1 Update priority 58
3.3.2.1.1 Update a ready task.....59
3.3.2.1.2 Update the running task.....60
3.3.2.1.3 Update group-rqs.....61
3.3.2.2 Schedule tasks.....62
3.3.3 Quantum 64
3.4 Lqsim simulator.....65
3.5 Summary.....68
Chapter 4 Demonstration of the Parasol CFS scheduler.....69
4.1 The basic CFS scheduler.....70
4.2 The group CFS scheduler.....71
4.2.1 Compare CFS with FCFS and PS 71
4.2.2 Single-processor and multi-processor.....74
4.2.3 Comparison of different share combination.....76
4.3 Mixed CFS scheduler.....77
4.4 Computation time.....78
4.5 Conclusion.....79
Chapter 5: Case Study.....80
5.1 The LQN model of the BBS.....81
5.2 Pre-experiment.....82
5.2.1 Part 1: Video component capacity 84
5.2.2 Part 2: User component capacity 85
5.3 Experiment 1: Using CFS group scheduling in AppCPU.....86
5.3.1 Experiment 1-1: Using PS scheduling.....86
5.3.2 Experiment 1-2: Applying CFS group scheduling.....89
5.3.3 Experiment 1-3: Different group share combination.....94
5.3.4 Experiment 1-4: Different group combinations.....96
5.4 Experiment 2: Apply the mixed CFS scheduler.....97
5.5 Experiment 3: Applying CFS scheduling to DB_CPU.....98
5.5.1 Experiment 3-1: Using PS scheduling to get the utilizations of two components.....100

5.5.2 Experiment 3-2: Applying the CFS group scheduler.....101

5.6 Conclusions.....103

Chapter 6 Conclusions.....105

References.....108

Appendix A: LQN core schema.....114

Appendix B: Grammar of the old style LQN input File.....134

List of Tables

Table 2.1 XMLSpy notations.....24
Table 3.1 The graphical notations of group element.....45
Table 4.1 Comparison of the basic CFS and PS scheduler (Pro2 is a single processor node) 70
Table 4.2 Comparison of the basic CFS and PS scheduler (Pro2 is a 2-processor node) 71
Table 4.3 The comparison of group CFS, FCFS and PS (TaskA and TaskB).....72
Table 4.4 The comparison of group CFS, FCFS and PS (UserA and UserB).....73
Table 4.5 Relationship among response time, multiplicity and group share.....74
Table 4.6 The comparison of a single-processor and 2-processor (TaskA and TaskB)...75
Table 4.7 The comparison of a single-processor and 2-processor (UserA and UserB)....75
Table 4.8 The comparison of different share combination.....76
Table 4.9 The mixed CFS scheduler (Pro2 is a single processor node).....78
Table 4.10 The mixed CFS scheduler (Pro2 is a 2-processor node).....78
Table 5.1 The Utilizations of the tasks in Processor AppCPU (the number of users is 1) 84
Table 5.2 The multiplicity parameters of tasks. 86
Table 5.3 Utilizations of tasks on the processor AppCPU.....89
Table 5.4 Group information 91
Table 5.5 Different Group share combinations (Number of users is 30).....94
Table 5.6 The utilization of user part and video part according to the DB tasks.....102

List of Figures

Figure 2.1 The structure of a red-black tree.....16
Figure 2.2 LQN Meta Model 17
Figure 2.3 LQN tasks and servers 18
Figure 2.4 Phases 20
Figure 2.5 LQN graphical notations.....21
Figure 2.6 The top level schema of LQN model.....23
Figure 2.7 Simplified PARASOL class diagram. 29
Figure 2.8 PARASOL task states and transitions.....30
Figure 2.9 PARASOL working process.....33
Figure 2.10 A Lqsim Multi-server task.....38
Figure 3.1 New LQN meta model (simplified) 44
Figure 3.2 The LQN input model with group definition. 46
Figure 3.3 The new LQN Core schema. 47
Figure 3.4 New parasol class diagram.....50
Figure 3.5 The structure of original ready-to-run-queue. 51
Figure 3.6 The structure of the host-rqs 51
Figure 3.7 The hierarchical structure of group-rq and host-rq 52
Figure 3.8 A Lqsim multi-task contained by a group 67
Figure 4.1 Basic Input model for testing.....69
Figure 4.2 Group input model (for testing) 72
Figure 5.1 Deployment of the Building Security System.....81
Figure 5.2 The Basic LQN model.....83
Figure 5.3 The response time of video part and user part of Pre-experiment1.....85
Figure 5.4 The Throughput of video part and user part of Pre-experiment1.....85
Figure 5.5 The response time of the user part of Experiment 1-1.....87
Figure 5.6 The throughput of the user part of Experiment 1-1.....87
Figure 5.7 The response time of the video part of Experiment 1-1.....88
Figure 5.8 The throughput of the video part of Experiment 1-1.....88

Figure 5.9 The utilizations of the User part and Video part of Experiment 1-1.....88
Figure 5.10 The basic group model 90
Figure 5.11 The response time of the user part comparing the CFS and PS scheduling.....92
Figure 5.12 Throughput of the user part comparing the CFS and PS scheduling.....92
Figure 5.13 The response time of the video part comparing the CFS and PS scheduling.....92
Figure 5.14 Throughput of the video part comparing the CFS and PS scheduling.....92
Figure 5.15 The utilizations of user part and video part in Experiment 1-2.....93
Figure 5.16 The response time of the group GetImg_T in the Experiment 1-3.....95
Figure 5.17 The response time of the group VidCtrl_T in the Experiment 1-3.....96
Figure 5.18 The response time of the group AcqProc_T in the Experiment 1-3.....96
Figure 5.19 The utilizations of case 1 in Experiment 1-4.....97
Figure 5.20 The utilizations of case 2in Experiment 1-4.....97
Figure 5.21 The response time of the user part comparing guarantee and cap share.....98
Figure 5.22 Utilizations of the user part and the video part in Experiment 2.....98
Figure 5.23 DB-model.....99
Figure 5.24 The Utilizations of the user part and video part in Experiment 3-1.....100
Figure 5.25 Utilizations of DB task of the user part and video part in Experiment 3-1.....100
Figure 5.26 The DB group model.....101
Figure 5.27 The Utilizations of the user part and video part in Experiment 3-2.....102
Figure 5.28 Utilizations of DB_Vid and DB_Info in Experiment 3-2.....102
Figure 5.29 The Utilizations of the user part and video part in Experiment 3-3.....103
Figure 5.30 Utilizations of DB_Vid and DB_Info in Experiment 3-3.....103

List of Listings

Listing 2.1 XML file layout.....23
Listing 2.2 Old Style LQN Input file.....25
Listing 3.1 An XML format LQN input model.....48
Listing 3.2 An LQN model in Old style LQN input file. 49

Glossary of Terms

BST	Binary Search Tree
BBS	Building Security System
CFS	Completely Fair Scheduler
FCFS	First Come First Serve
FQ	Fair queuing
GPS	Generalized Processor Sharing
HOL	Head of Line
LQN	Layered Queueing Network
MOL	Method of Layers
PPR	Priority Preemptive Resume
PS	Processor Sharing
QoS	Quality of Service
RAND	Random scheduling
SFS	Surplus Fair Scheduling
SPE	System Performance Engineering
SRVN	Stochastic Rendezvous Network
WFQ	Weighted Fair Queuing

Chapter 1 Introduction

The behavior of the scheduler of an operating system has never lost the attention of researchers. Today, multi-processors and multi-threaded techniques enhance the importance of schedulers. Before the 1980's, traditional CPU schedulers tried to fairly allocate resources among processes. Processes or tasks in different groups or belonging to different users were not treated differently according to their rights. With the development of networks and the wide use of multi-user systems, the fair share scheduler [16] was developed and was designed for allocating resources fairly among the users or organizations.

One of important purposes of schedulers is to ensure fairness corresponding to the rights amongst the parties utilizing the resources, and has become a requirement of Quality of Service (QoS). More and more applications are adopting the fair share scheduler and many researchers are now focusing their attention on this field [1, 15, 20, 30].

HP, IBM, and Sun provide the FS scheduler package on their UNIX platforms [5]. Linux too has adopted a fair share scheduler as of the version 2.6.23 of the kernel. The previous O(1) scheduler [14] was changed to a new fair share scheduler, called Completely Fair Scheduler (CFS) [18], even though the CFS scheduler (O(log n)) is slower than the

previous one. The CFS scheduler has become the default scheduler. Therefore, CFS is the main focus of this thesis work.

Analyzing the performance of systems is drawing more attention because performance problems lead to cost overruns or project cancellations [34]. Software Performance Engineering (SPE) [34] is the process of building a responsive software system. The objective of SPE is to analyze performance models of a system throughout its lifetime. Performance prediction and evaluation in the early design phases are essential to software application development. Early performance prediction can start from the very beginning, the phase which the system is still accessed for feasibility. Performance analysis attempts to locate potential bottlenecks and other performance constraints or limitations as early as possible and provide feedback for the current design of the system. Locating performance problems early can substantially reduce the risk of project delays or failures caused by performance problems. This can remarkably reduce the development cost.

1.1 Motivation

Layered Queueing Networks (LQNs) are recognized as an important performance analysis model [34]. The LQN performance model is designed for modeling distributed systems with a client-server architecture. It can predict software performance by finding potential bottlenecks.

LQN analytic and simulation tools support several scheduling types, such as First Come First Serve (FCFS), Priority Preemptive Resume (PRR), Head Of Line (HOL), and

Processor Sharing (PS). Since fair share scheduling is becoming more widely used, the impact of fair share scheduling on the performance of software applications is attracting more interest. It is essential for LQN performance tools to support fair share scheduling and find rules to predict the performance of such software applications. The goal of this thesis is to integrate the CFS scheduling policy into the LQN performance model.

This thesis work will focus on how to apply the CFS scheduler to the LQN model. To do so, it will extend PARASOL, a discrete event simulator, and Lqsim, the Layered Queueing Network simulator which uses PARASOL as its simulation engine, to support CFS scheduling. Finally, this thesis will then explore how the CFS scheduler can affect the performance of an example system.

1.2 Contributions

1. Integrating the CFS scheduling policy into the LQN model, and making extensions to the LQN model components.

2. Developing the PARASOL CFS scheduler to extend the original built-in scheduler with three features: basic CFS scheduling, group CFS scheduling and mixed CFS scheduling.

3. Extending the simulator Lqsim to incorporate the CFS scheduling policy.

4. Exploring the performance impact of the CFS scheduler by a case study and providing the heuristics for the use of CFS scheduling.

1.3 Thesis Organization

The thesis consists of six chapters. Chapter 2 presents the necessary background information for the thesis. First, it provides a more detailed introduction to the concepts related to the fair share scheduling and a brief literature review of fair share scheduling. Further, it presents the principles of the CFS scheduling policy and the Linux CFS scheduler. Next, the components of the LQN model are described. Finally, the simulation system of the LQN tools is introduced, and a brief introduction of event-driven simulation is presented.

In Chapter 3, the design of the CFS scheduler for LQN is presented in detail. In the first part, a detailed description is presented about how to apply the new component to the LQN model. The second part, also the most important part, describes the structure of the CFS ready-to-run-queue and the implementation of the CFS scheduler. The final part describes the extension to the simulator, Lqsim, to support the CFS scheduler.

In Chapter 4, several tests are performed to verify the PARASOL CFS scheduler. Comparisons are made to the PS and FCFS scheduler.

In Chapter 5, a case study of a building security system (BSS) is described in detail. The chapter states a possible performance problem the system may have, and applies the CFS scheduler to one of the processing nodes, to try to remove the problem and improve the system performance.

In Chapter 6, conclusions and limitations are provided.

Chapter 2 Background

This chapter describes the necessary background information required to understand the thesis. The first part introduces fair share scheduling in general and the Linux CFS scheduler in particular. The second part of this chapter demonstrates the structure of the LQN model. Finally, the operations of the LQN simulation tool Lqsim and its underlying simulation engine PARASOL are described.

2.1 Fair share scheduling

The fair share scheduler [16] has become a widely used scheduler in the last two decades. Fair share scheduling was originally designed for managed resource allocation of processes on a timesharing uni-processor system [16]. In the early scheduler, the behavior of the scheduler tried to treat all processes identically. The consequence is that users with more processes could get more CPU resources than the users with fewer processes. If there were very large number of processes running in the system, then all users would suffer poor response time. To improve upon this situation, the paper of Kay and Lauder [16], describes the concept of the fair share scheduler through the implementation of charge management of a student lab environment.

2.1.1 Share

The fair share scheduler will adjust the scheduling priority of a process according to the share the process is entitled to.

The term share is used to define the user's entitlement for resources consumption. The greater the share, the more resources a user can get from the machine. The shares are usually determined by administrators and may change over time [16]. The utilization of a user is the amount of the work the user has done per unit of time. A fair share scheduler tries to keep the usage of a user to the share of resources to which the user is entitled. To achieve this, the general method is that the scheduler lowers the priority of the current running process, and increases the priorities of all other processes.

when assigning a small share to a large process, the process can finish eventually, but this will slow it down. This gives the idea as to how to use the fair share scheduler to impact and tune the system performance. Simply stated, a share indicates the relative importance of processes.

Using the same principle, fair share can apply to "groups". Processes or users can be collected into several groups as needed. Group shares are the values representing for group priorities in a fair share system. The users in the same group will get the same portion of the group share. The larger the number of active processes in a group, the fewer portions each process in the group gets. The number of processes in one group will not affect the performance of processes in other groups. Groups with larger shares have a higher priority.

For example, if there are three groups (A, B, C) with shares of 30%, 20%, and 50%, and containing two, three, and four users respectively, the available CPU resource will be partitioned as follows:

Group A: (30% / 2 users) = 15% per user

Group B: (20% / 3 users) = 6.7% per user

Group C: (50% / 4 users) = 12.5% per user

If one user in group A is executing 2 processes, and the other user has 5 processes running, the CPU resource will be partitioned as follows:

Group A: User 1: (15% / 2 processes) = 7.5% per process

Group A: User 2: (15% / 5 processes) = 3% per process

If more processes become active in a group, each process within the group will get less CPU resource.

2.1.2 Guarantee and Cap

Guarantee and Cap [5] describe two kinds of shares. Guarantee is used to define the minimum share which a user or group should get. Guarantee means a process or a group can get more CPU resources if there is a surplus; its utilization may be greater than its share. Cap is defined as the maximum share which a process or group should get. The utilization of a group can not exceed its share. Processes will have poor response times if their utilizations approach their shares.

Using the previous example, suppose that there is no active process in group B. If the group shares are defined by guarantee, group A and group C may get 10% more CPU

time each. If the group shares are defined by a cap, group A and C still get 30% and 50%, i.e. no more than their shares.

2.1.3 Types of fair share scheduling

There are several scheduling types related to fair share scheduling.

2.1.3.1 Fair queuing

Fair queuing (FQ) [20] is a scheduling scheme used in computer networks and statistical multiplexing to allow several data flows to fairly share the link capacity [20]. It has become one of the most popular scheduling methods. The general idea is the resource, for example a buffer, is divided into many queues. The data packets are stored temporarily in queues waiting to be forwarded. The packet with the minimum finishing time is forwarded first. FQ achieves max-min fairness which means the highest priority is given to the minimum data rate of any active data flow. FQ is applied frequently in routers, switches and multiplexors. FQ avoids scheduling starvation.

2.1.3.2 Weighted Fair Queuing

Weighted Fair Queuing (WFQ) is a data packet scheduling technique allowing different scheduling priorities to statistically multiplex data flows [27]. WFQ is a generalization of Fair Queuing (FQ). If all queues have the same weight, WFQ becomes FQ. The general idea is WFQ puts the incoming packets into some separate flow queues and sends out a fixed portion for each flow at a time. As with FQ, each data flow has its own FIFO queue in WFQ. Several active data flows are served simultaneously. Contrary to FQ, the

different sessions can be assigned to have different service shares in WFQ. According to the share, each flow queue is entitled to a different weight. Short queues can finish first. One advantage of WFQ is that this policy can prevent the starvation of a short queue. WFQ is usually used in communication networks and multiplexing. In [30], WFQ becomes an approach to control the Quality of Service, by regulating the WFQ weights dynamically.

2.1.3.3 Proportional-share scheduling

Proportional-share scheduling [28] is a scheduling policy when the clients competing for resources will receive a portion of resources in proportion to their ticket allocations. Proportional-share scheduling can be implemented either by the lottery scheduling or the stride scheduling algorithm.

Lottery scheduling is a randomized resource allocation mechanism [24, 29] that efficiently implements proportional-share resource management. The general idea is that all the resource rights are represented by the lottery tickets. Every process has a certain number of tickets. After a process wins in a lottery, the resource will be allocated to this process. The chance of winning in the lottery is a proportion to the number of tickets that the process holds, so granting a process more tickets provides it with a relatively higher chance of selection.

Lottery scheduling avoids the starvation problem. Since each process holds a non-zero number of tickets, there exists a non-zero probability that guarantees the process will win a lottery eventually.

Stride scheduling is a deterministic allocation algorithm. Similar to lottery scheduling, resource rights are represented by tickets, and resources are allocated to competing clients in proportion to the number of tickets held [2]. The general idea is that each client has a time interval, called a stride, which determines how often the client is scheduled. The length of a stride is inversely related to the number of tickets held. For example, a client with twice the number of tickets than another will have half the stride and will be allocated twice as frequently when compare to the others. The main difference with lottery scheduling is that lottery scheduling is probabilistically fair, whereas stride scheduling is deterministically fair.

There is a concept called generalized processor sharing (GPS) [23] worth some attention. GPS is an idealized algorithm where each application has a weight and all applications are allocated a processor bandwidth in proportion to their weights. GPS can guarantee fairness in a single-processor environment. GPS-based fair scheduling algorithms are unsuitable for fair allocation of resources in multiprocessor environments [7] because it will cause unfairness or starvation. WFQ and proportional-share scheduling are examples of GPS-based scheduling algorithms.

2.1.4 Related works to fair share scheduling

Fair share scheduling opened a new research field after it was presented in the early 1980's. Early studies on fair share scheduling were presented in [16, 32]. [32] presented a feedback system to realize fairly sharing resources, with performance constraints. In [16], the fair share scheduler was applied in a single processor environment and the shares of users are caps.

In recent years, many researchers have focused on the allocation of processor capacity and network packet scheduling. Most of the types of fair share scheduling described in the previous section came from these fields, as well as their extensions [2, 4, 15].

This research has resulted in some other proposed fair share scheduling algorithms. [1] presented a vacation model which is applied to a polling system with a fair share scheduler. In [9], a decentralized architecture is presented for a Grid-wide fair share scheduling system. [7] presented the surplus fair scheduling (SFS) and implemented SFS in the Linux kernel. SFS is a proportional-share scheduler which is designed for multiprocessor environments, and it is a generalized multiprocessor sharing algorithm.

Many operating systems support fair share scheduling. For example, [39] describes the principle and usage of the fair share scheduling in Solaris 9. The Sun Solaris 9 operating system adopts the fair share scheduler as a regular scheduler. Users can configure the fair share scheduler through the command line. The fair share scheduler treats processor sets as independent units, and the allocation of CPU resource is controlled individually by the fair share scheduler. Linux adopts CFS as the basic scheduler, which makes fair share scheduling play an important role in operating systems.

Another research trend in fair share scheduling concerns the impact of fair share scheduling on performance model and performance models were studied intensively in [2, 4, 5, 17, 32].

In [5], a model for transaction workloads to fairly share CPU resources was developed. Furthermore, the impacts of the fair share scheduler on the system performance were

demonstrated. Based on the work of [5], some refined algorithms were presented to improve the efficiency of fair share scheduler in large systems in [4]. In [17], the authors discussed the effectiveness of a fair share scheduler on a high performance computing system. The conclusion is that the fair share scheduler does not play a big role in altering important performance metrics. Fairness on a uni-processor can be achieved, in principle, over all time periods because the processes can be swapped in and out freely according to the frequency that the fair share algorithm updates. However, cluster processes must run on a specific number of processors and run to completion on those processors once started.

2.2 Completely fair scheduling

As described by the author of CFS, Ingo Molnar, the CFS is for "basically [modeling] an 'ideal, precise, multi-tasking CPU' on real hardware."[40] The "ideal, precise, multi-tasking CPU" is a CPU which can run multiple processes simultaneously. Each process running on this ideal CPU will get an identical portion of processor's time. For example, if two processes are running in parallel, each will have one half of the processor time. On real hardware, only a single process can run on the processor at one time, while the other processes are waiting for the CPU. So the current running process consumes more CPU time than it should have. This results in an imbalance of allocation of CPU resources. Meanwhile, the waiting processes become grave in need for requiring the CPU resource than the running process. The Completely Fair Scheduler tries to run the task with the "gravest need" for CPU time [18]. The CFS scheduler tracks and minimizes the

unfairness of each process after a single execution and makes sure every process get its fair share of CPU resource.

The nature of the CFS scheduler is that it executes each process for a very short time period, and switches quickly to another ready process. The aim of basic CFS scheduling is to provide an illusion of concurrency.

CFS scheduler is a priority-based scheduler. The general idea of CFS scheduler is: when a process is running on a processor in a given time period, it will get more CPU time than it is entitled to. The overrunning time will decrease the priority of the process. Conversely, when a process is waiting for the CPU, the time it would run on the ideal CPU will become a credit for its waiting time and will then increase its priority as well. In CFS, the term 'fair' is used for representing the priority of the processes. The greater the value of fair, the higher is the priority, and the graver the need for CPU. Ideally, if the CFS scheduler ensures the fairness to each process, its fair value should approach zero. The scheduler will track each process's fair, rank processes by their fair, and then determine which task should run next and how long the task should execute before it is preempted.

CFS scheduling consists of two levels of scheduling: basic CFS scheduling and group CFS scheduling.

2.2.1 CFS group scheduling

The idea of CFS group scheduling is:

1. Collect sets of processes into groups, and assign a share to each group.

2. The CFS group scheduler will consider the priority of the groups first, and decide which group has the right to run next.

3. The scheduler will pick up one process in the highest priority group and run it.

Consider an example with two groups, A and B, which are running processes on a machine. Group A has just two processes running, while group B has ten processes running. There are two cases that should be considered. If there is no share specified, groups A and B both get a 50-50 share. The processes in group A will get 25% CPU each, while the processes in group B will get 5% CPU each. If each group has a share, say, 30% and 70% for groups A and B, the processes in groups A and B will get 15% and 7% each respectively. Therefore, the CFS group scheduler is fair to groups A and B, rather than fair to all 12 processes in the system.

For the first case, if more groups are defined, the share of each group will decrease. In the second case, each group can get a portion of CPU which is greater or at least equal to its share.

2.2.2 The Linux CFS scheduler

The Linux CFS scheduler replaced the O(1) Scheduler [14] in the version of 2.6.23 Linux kernel release. The running time of the CFS scheduler is in O(log n), and it provides fair interactive response.

In Linux, real time processes have higher priorities and have a separate run queue. The non-real time processes are scheduled by the CFS scheduler in the time not used by real-time processes. The Linux CFS scheduler completely ignores sleeping time, interactive process identification, and time slices, which are the parameters used by earlier schedulers [18]. The Linux CFS scheduler ranks the priority of processes by the fair value. A short sleeper might be entitled to some bonus time to increase its fair value, to make sure it wakes up and responds quickly.

In the Linux CFS scheduler, granularity describes how quick the scheduler will switch processes in order to maintain fairness, and it is the only tunable variable. A low granularity causes more frequent switching and lowers the throughput, but it provides quicker interactive responses.

2.2.3 Red-Black Trees

The ready-to-run-queue of the CFS scheduler is implemented as a red-black tree. A red-black tree [8, 31] is a type of self-balancing binary search tree (BST). It is more complex than other BSTs, but it guarantees a worst-case execution time for its most operations. It can search, insert, and delete in O(log n) time, where n is the number of elements in the tree.

The structure of a red-black tree is illustrated in Figure 2.1. In addition to the features of an ordinary BST, a valid red-black tree must satisfy the following requirements: [31]

1. A node is either red or black.
2. The root must be black.

3. All leaves are black, and contain no data (NIL).
4. Both children of every red node are black.
5. Every simple path from a node to a descendant leaf contains the same number of black nodes, either counting or not counting null black nodes.

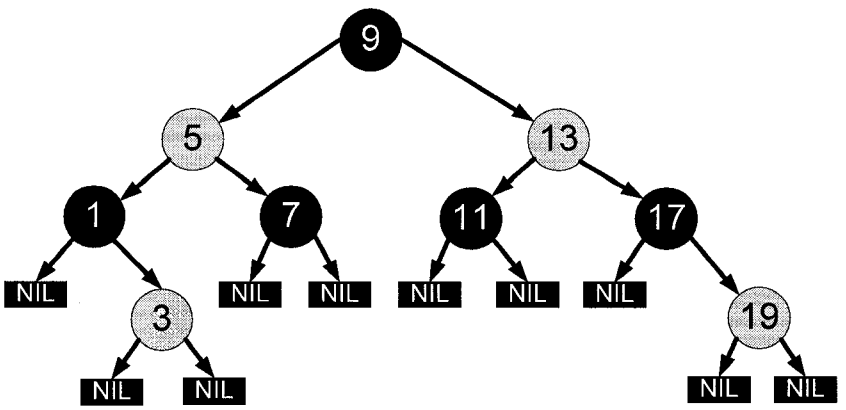


Figure 2.1 The structure of a red-black tree.

All of these features make a red-black tree a balanced tree. As a consequence, it provides better worst-case execution time than other BSTs, so the red-black tree is very efficient in practice. For example, there are many uses of red-black trees in the Linux kernel.

2.3 Layered Queuing Network (LQN)

The LQN model is used to model software systems and evaluate their performance. A Layered Queueing Network (LQN) [10, 13, 35] is an extended regular Queueing Network, and is used to evaluate system with a layered structure [12], such as are found in many distributed computer systems today. The layered structure arises because a server in one layer may make requests to other servers in lower layers. The LQN model is the extension of both the Stochastic Rendezvous Networks (SRVN) [35] and the Method of Layers (MOL) [26].

2.3.1 Elements in LQN model

The LQN meta-model is shown in Figure 2.2. There are six essential elements for modeling software systems: processors, tasks, entries, activities, precedence and requests.

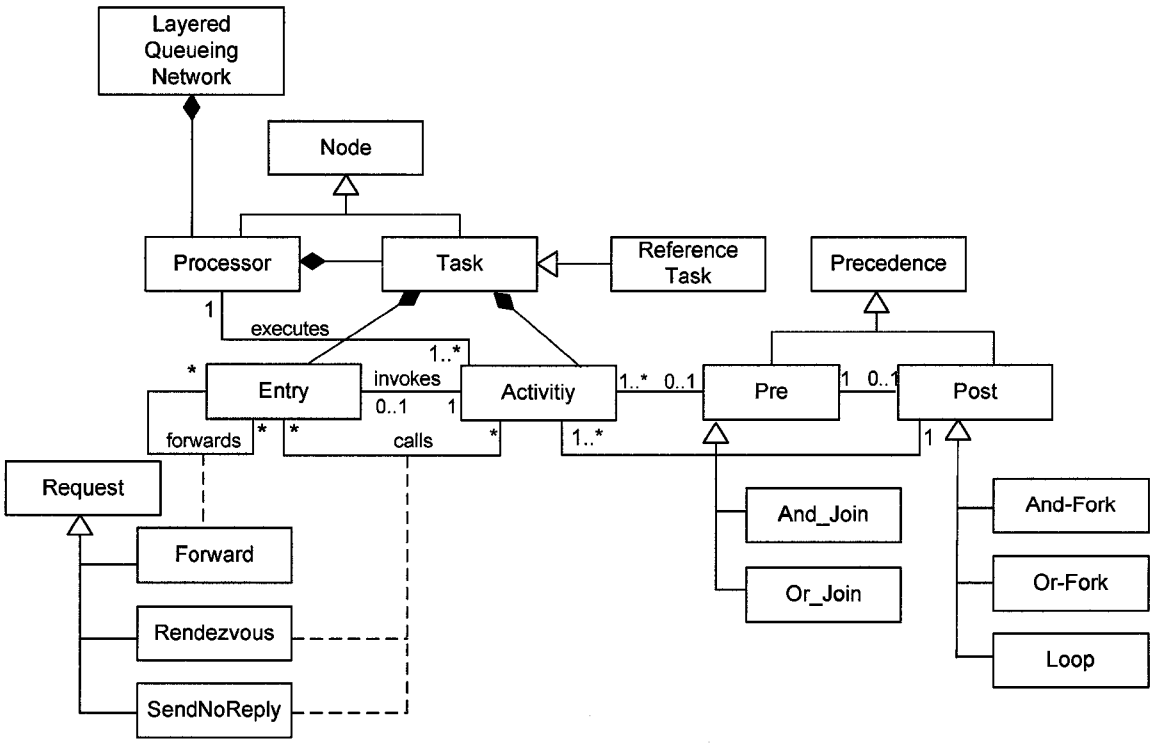


Figure 2.2 LQN Meta Model [13]

2.3.1.1 Processors

In the LQN model, processors are used to consume time. They are pure servers because they only accept requests. Every processor has a single queue for incoming requests, and the queueing disciplines include FCFS, PPR, HOL, PS, and RAND (Random scheduling). A processor can be a single processor or multiprocessor or infinite processor; the latter is used to represent a pure delay. Processors are often used to model physical CPUs.

2.3.1.2 Tasks

Tasks can model either software or hardware resources. A LQN task has a single queue for incoming requests, and the queueing discipline can be one of FCFS or HOL. Tasks run on processors to handle the requests. Tasks may make requests to other server tasks. Example LQN tasks are shown in Figure 2.3.

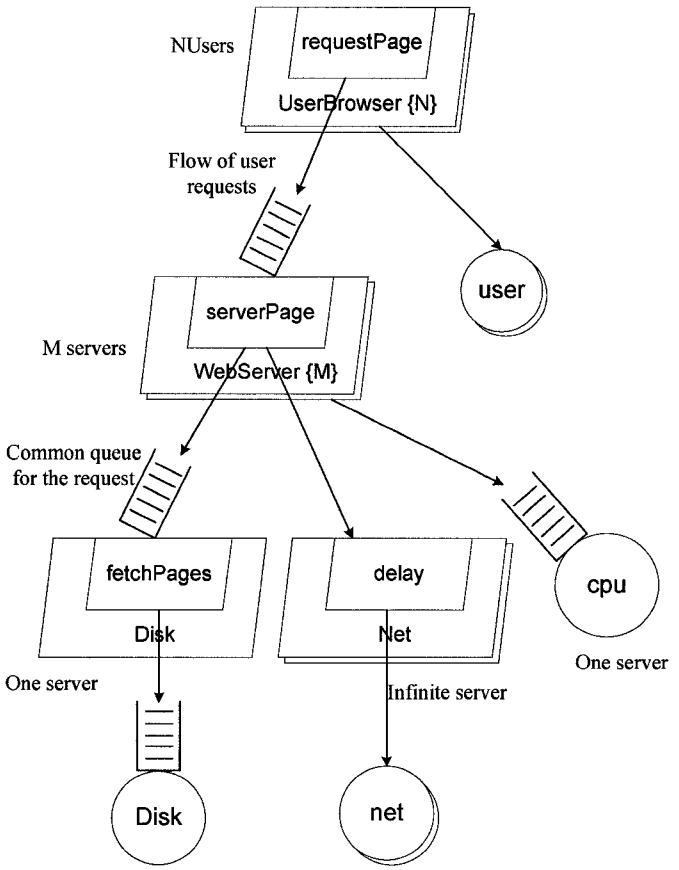


Figure 2.3 LQN tasks and servers [33]

A LQN Task can be either a reference task or a server task. A reference task is a task which sends requests only, so it is a pure client. It is used to model customers in the model, and is used in closed models to generate the load in the system. In Figure 2.3, the UserBrowser task is a reference task.

A server task accepts requests from the other tasks and can send replies back. A server task can be a single server task, a multi-server task or an infinite server task depending on the number of homogeneous threads able to service requests. An infinite server can create an infinite number of threads, so requests will never queue at this type of task. Infinite servers can run on an infinite processor and are often used to model a pure delay, such as network delay. For a multi-server, all the threads of the multi-server share the same queue. In Figure 2.3, task WebServer is a multi-server task, task Disk is a server task, and task Net is an infinite server task.

2.3.1.3 Entries

Entries model the service offered by tasks. A task may contain one or more entries, and these entries share the same queue. An LQN entry is used to serve the request, which can be either synchronous or asynchronous. If the client makes a synchronous request, the entry generates a reply and usually sends it back to the client. Alternatively, an entry can forward the reply to a subsequent server instead of replying to the client. The reply becomes a synchronous request to the receiving server. The reply to a forwarded request is sent to the original client.

The demands of an entry can be specified through phases. The first phase is a service phase, where the sender is blocked. This phase ends after the reply is sent. Subsequent phases are used for concurrent execution in that the client and server can both execute in parallel. Figure 2.4 illustrates the principle of phases. The demands of entries can be specified by LQN activities which are described next.

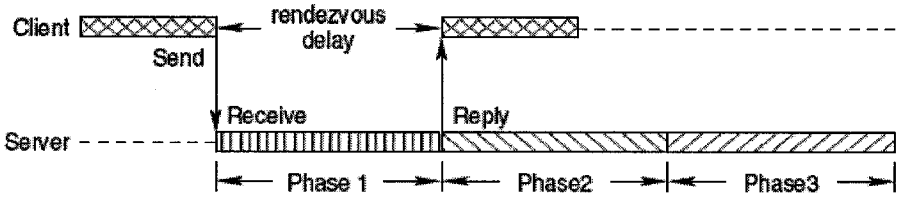


Figure 2.4 Phases [14]

2.3.1.4 Activities

Activities are the lowest-level of specification in the performance model [12]. Activities are used to specify more complex interactions. They are connected to each other by precedence elements. Activities make requests to entries on other tasks and generate replies sent to the client. Activities consume their service time on processors.

2.3.1.5 Precedence

Precedence is used to connect activities in an activity graph. Further details about precedence and activities can be found in [12, 35].

2.3.1.6 Requests

There are three types of requests: rendezvous, forwarded, and send-no-reply. A rendezvous request is a synchronous request. The client is blocked until the server replies to the request. A send-no-reply request is an asynchronous request. The client will continue its own execution after sending a request; no reply is expected. A forwarded request results in the request being redirected to a subsequent server which will reply to the original client [12].

2.3.2 Graphical Notation of LQN

The LQN model can be expressed by a set of graphical notations. Figure 2.5 shows an example, which is the simplified version of the model in the case study described in Chapter 5.

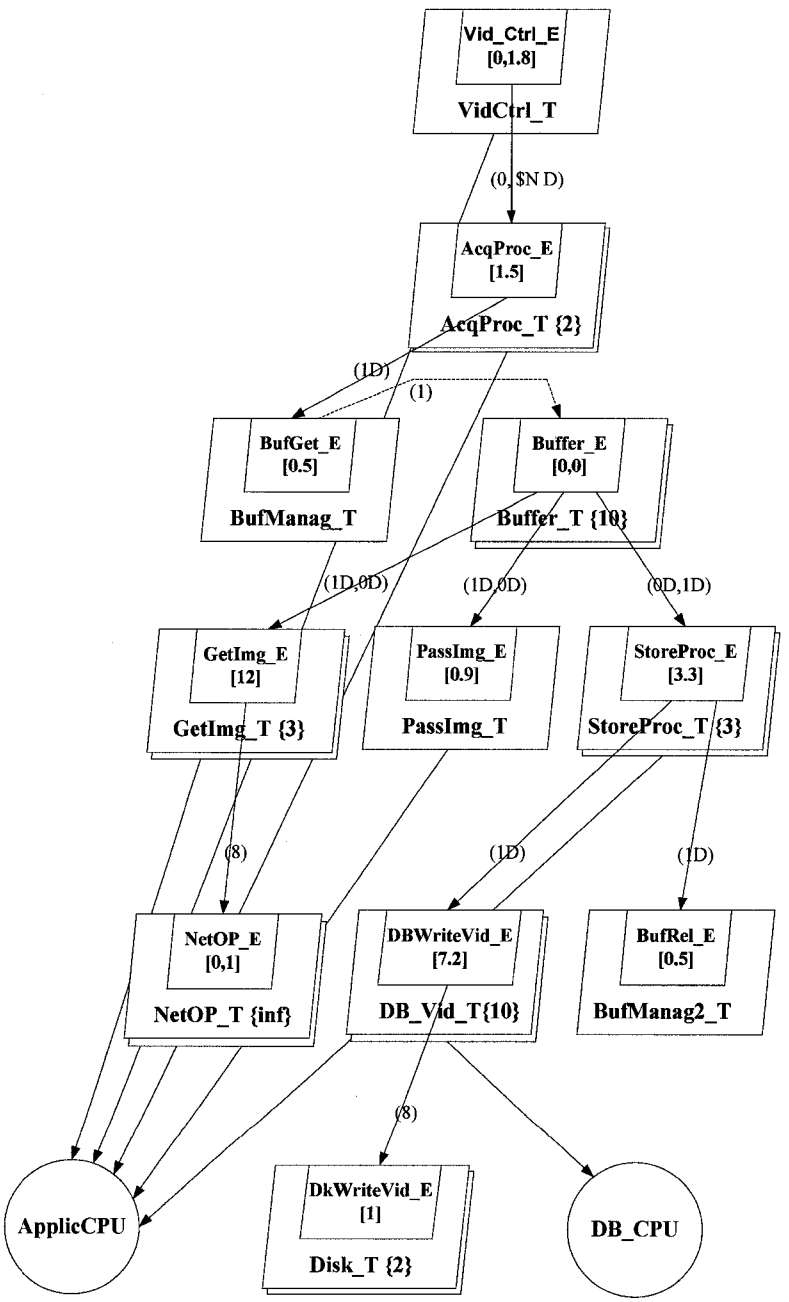


Figure 2.5 LQN Graphical Notations

Processors are represented using circles with arcs connected to the tasks which they run. Tasks are represented by parallelograms, and a double-parallelogram stands for a multi-server task. The integer in a pair of braces is the multiplicity of a task. Entries are shown using the small parallelograms inside of the task icons. The label inside of a pair of square brackets is the service demand of the entry specified by phase. An arc with a solid arrow head stands for a synchronous request, while one with an empty arrow head stands for an asynchronous request, which is not present in this figure. The dashed line with a solid arrow head represents a forwarding request. The label for an arc is the mean number of requests from a phase to an entry.

2.3.3 XML Schema of LQN Component

A LQN model can be represented using either an XML file (described here) or a text file (described in the next section). There are three basic component models in the XML format; they are the assembly model, the sub-model and the core model respectively [36]. Of these three models, only the core model is of concern here as it specifies all the elements shown in the LQN Meta model in Figure 2.2. In the thesis, modifications will be made to the LQN core model only. The other two models are concerned with the assembly of a LQN model from component sub-models and are not discussed further here. Refer to [36] for more information.

The LQN model is defined by lqn.xsd, it is the root of the schema. The LQN model is defined by the LqnModelType, which consists of run-control, plot-control, solver params, an LQN-core type. LqnModelType is shown in Figure 2.6.

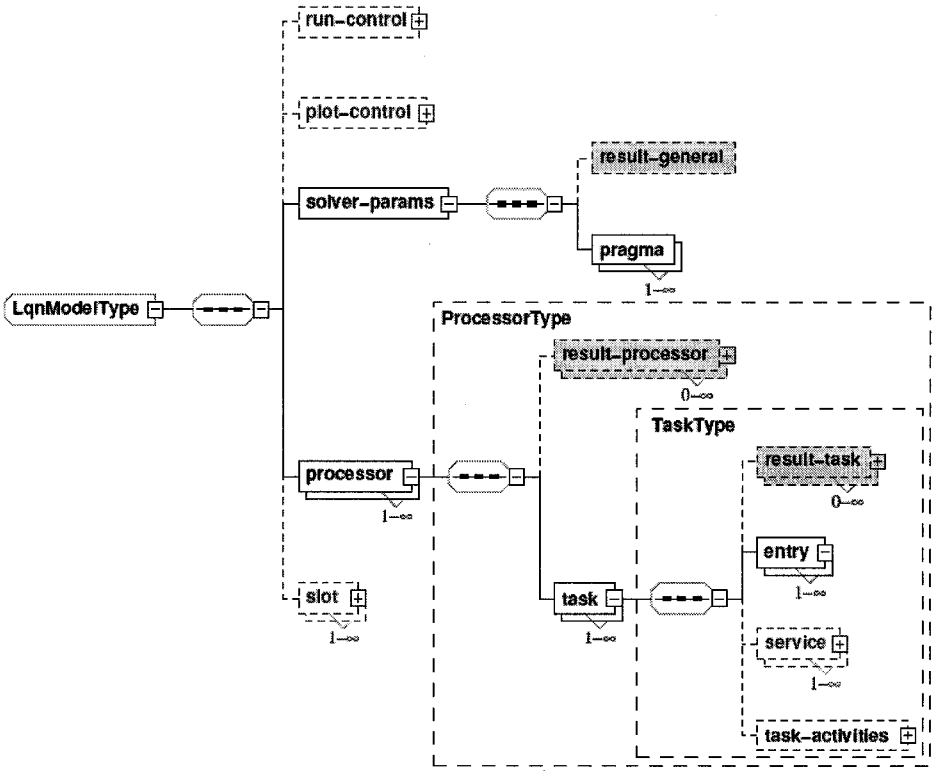


Figure 2.6 The top level schema of LQN model [12]

Figure 2.6 shows the LQN core model as well. The lqn-core type contains one or more processors and may have a slot. The slot defines the interface and binding facilities [36] used when assembling models. Processors are defined by ProcessorType, contain one or more tasks and may have a result-processor. The attributes of ProcessorType include speed, multiplicity, scheduling type, etc. Tasks are defined by TaskType, which consists of four elements: entries, activities, services and result-tasks. The last three elements are optional elements. Listing 2.1 shows the layout of the LQN XML format input file.

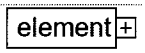
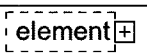
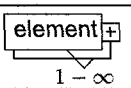
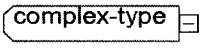
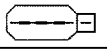
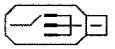
Listing 2.1 XML file layout [12]

```
<lqn-model>
    <solver-params>
      <pragma/>
   </solver-params>
```

```
    <processor>
        <task>
           <entry>
                <entry-phase-activities>
                     <activity>
                          <synch-call/>
                          <asynch-call/>
                     </activity>
                     <activity> ... </activity>
                </entry-phase-activities>
            </entry>
           <entry> ... </entry>
           <task-activities>
                <activity/>
                <precedence/>
            </task-activities>
        </task>
        <task> ... </task>
     </processor>
     <processor> ... </processor>
</lqn-model>
```

The schema is presented by XMLSpy [38] notations, which are shown in Table 2.1. Boxes represent elements in the schema. Stacked boxes represent multiple instances of the elements. Dashed lines and boxes represent optional components. Octagons represent sequences and choices.

Table 2.1 XMLSpy notations

Name	Icon	Type	Description
Element		Component	An element. The + denotes that the element is expanded else where
Element		Component	An optional element
Multiple Element		Component	A sequence of the named element.
Complex Type		Component	A complex type element may contain other elements.
Sequence		Compositors	A sequence of elements
Choice		Compositors	A choice of elements

2.3.4 Old Style LQN Input Model

The old style LQN input file format consists of five sections, which specify the solver parameters, processors, tasks, entries and activities respectively. One major difference to notice is that elements are grouped by type (e.g., task, processors) in the old style input format, whereas they are nested in the XML input format. Listing 2.2 shows an example of this input file format. The example is from [36].

Listing 2.2 Old Style LQN Input file

```
G #General information section
"This is a test case" #comments of the model
0.00001 #convergence criterion
50 #iteration limit
1 #print interval
0.8 #under-relaxation
-1 #end of General section

# Processor Information section
P 0
# SYNTAX: p ProcessorName SchedulingDiscipline [multiplicity,
default = 1]
#SchedulingDiscipline = f FCFS(FIFO)|r RAND|p PPR|h HOL|s PS
#multiplicity = m integer (multiprocessor)|i (infinite
processor)
p Pro1 f #Processor Pro1 (FIFO)
p Pro2 s 0.1 #Processor Pro2 (PS)
p Pro3 f #Processor Pro3 (FIFO)
-1 #end of Processor section
# Task Information section
T 0
#SYNTAX: t TaskName taskTypeFlag EntryList -1 ProcessorName
[multiplicity]
#taskTypeFlag = r (reference task)|n (normal task)
#multiplicity = m integer (multithreaded)| i (infinite task)
t UIF r user -1 Pro1 m 10
t TaskA n e1 e2 -1 Pro2
t TaskB n e3 e4 -1 Pro2
t TaskC n e5 -1 Pro2
t TaskD n e6 -1 Pro3 m 10
-1
#Entry Information section
E 0
# SYNTAX: Token EntryName Value1 [Value2] [Value3] -1
```

```
s user 1.0 -1
y user e1 5 -1
y user e2 10 -1
s e1 0.04 -1
y e1 e3 3 -1
s e2 0.2 -1
y e2 e4 2 -1
s e3 0.01 -1
y e3 e5 1 -1
s e4 0.05 -1
y e4 e5 3 -1
s e5 0.7 -1
y e5 e6 3 -1
s e6 0.05 -1
-1
```

2.4 LQN simulation tools

There are two main approaches for simulating the behavior of a software system: time-slicing simulation [25] and event-driven simulation [3]. The difference between the two methods is the way the simulation time is advanced. Time-slicing simulation, also known as exhaustive simulation [19] can process events at every clock cycle, although there can be no change of system states in some cycles. Event-driven simulation advances the simulation time in irregular intervals. Compared to time-slicing simulation, event-driven simulation can save considerable processing overhead.

2.4.1 Components of a Discrete Event driven Simulation

In a discrete event driven simulation [3] system, discrete events represent the change of the value of a state in the system [25]. Each event is treated as an instant in time regardless of the computation time needed by the real case.

The simulation is driven by an event list which is a list for events waiting to occur. The event list stores events in chronological order. When an event is inserted, it has a time stamp indicating the time the event will happen, and a value indicating the type of the event. The first element in the list will always be the next event to be executed. The type of the event determines the actions the simulator takes at the time the event executes. This may result in the addition or removal of events to or from the event list. The events in the event list do not necessarily have same type.

Another important element in the simulator is the clock. A clock is used for tracking the simulation time by the simulator. In discrete-event simulations, the clock will skip the time period between two events; simulation time will jump to the time of the next event.

As an event occurs, simulation driver will handle it and collect the statistics of system variables at same time. Here is a generic event-driven algorithm for simulation drivers:

1. Initialize system state
2. Initialize event list
3. Initialize Clock (usually starts at simulation time zero).
4. While (Stopping condition is FALSE)
    a. Collect statistics from current state
    b. Remove first event from list, handle it
    c. Set time to the time of this event.
5. Generate statistical report.

The stopping condition can be certain number of iterations of the loop, a certain value of a system variable or state, or a time out event.

To summarize, the event-driven simulation engine continuously takes the first event from the event list. The simulation engine handles each event in order of increasing time. Execution continues until all events have been processed or reach the stopping condition. The simulation driver can be single-threaded or multi-threaded.

2.4.2 Parasol Simulation Engine

PARASOL [21, 22] is an event-execution-based simulator and it provides a prototype of a distributed software environment. It was developed by Prof. John E. Neilson from the Computer Science Department of Carleton University. PARASOL is a single process simulation tool. But like a multitasking operating system, PARASOL gives the illusion of concurrent execution of multiple tasks by using its own threads. PARASOL serves as a Software Performance Engineering (SPE) tool spanning the software development process from early design through to system maintenance and tuning [21].

PARASOL is designed to simulate distributed and parallel computer system models [21]. Figure 2.7 shows a simplified class diagram of PARASOL entities. Since only processing nodes, hosts, tasks are concerned in this thesis, Figure 2.7 omits some entities, such as links, buses and statistics.

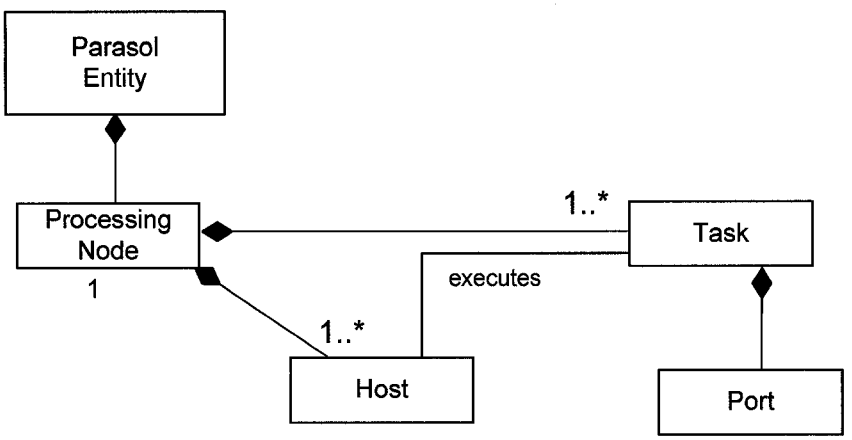


Figure 2.7 Simplified PARASOL class diagram

First, PARASOL can simulate the hardware environment of a distributed system. The construction of networks includes building processing nodes and their communication facilities, such as links and buses. Each processing node may have one or more processors, called hosts. Tasks may be allowed to be executed on a specified host only or on any of them.

User activities are accomplished using PARASOL tasks. The task management functions can perform the creation, activation, alteration of states, and task destruction. The states and transitions of tasks are shown in Figure 2.8. This thesis will focus primarily on those states involved with scheduling, namely READY, HOT, and COMPUTING. When a task is able to execute, for example when it receives a message, it is moved to the READY state. At some point, it will be scheduled to run on a host which will cause it to go into one of the executing states.

Several tasks may be in executing states simultaneously to give the appearance of concurrent execution. However, the PARASOL simulator can run only one thread at one time. The task which is executing this thread is labeled as being in the HOT state. User-

defined simulation code is executed at this time and will always cause an outgoing transition through lock, hold, sleep, compute, sych or kill. The other running tasks remain in some other executing states, such as COMPUTING, SYNCING, and SPINNING state, to represent that they are running on different hosts.

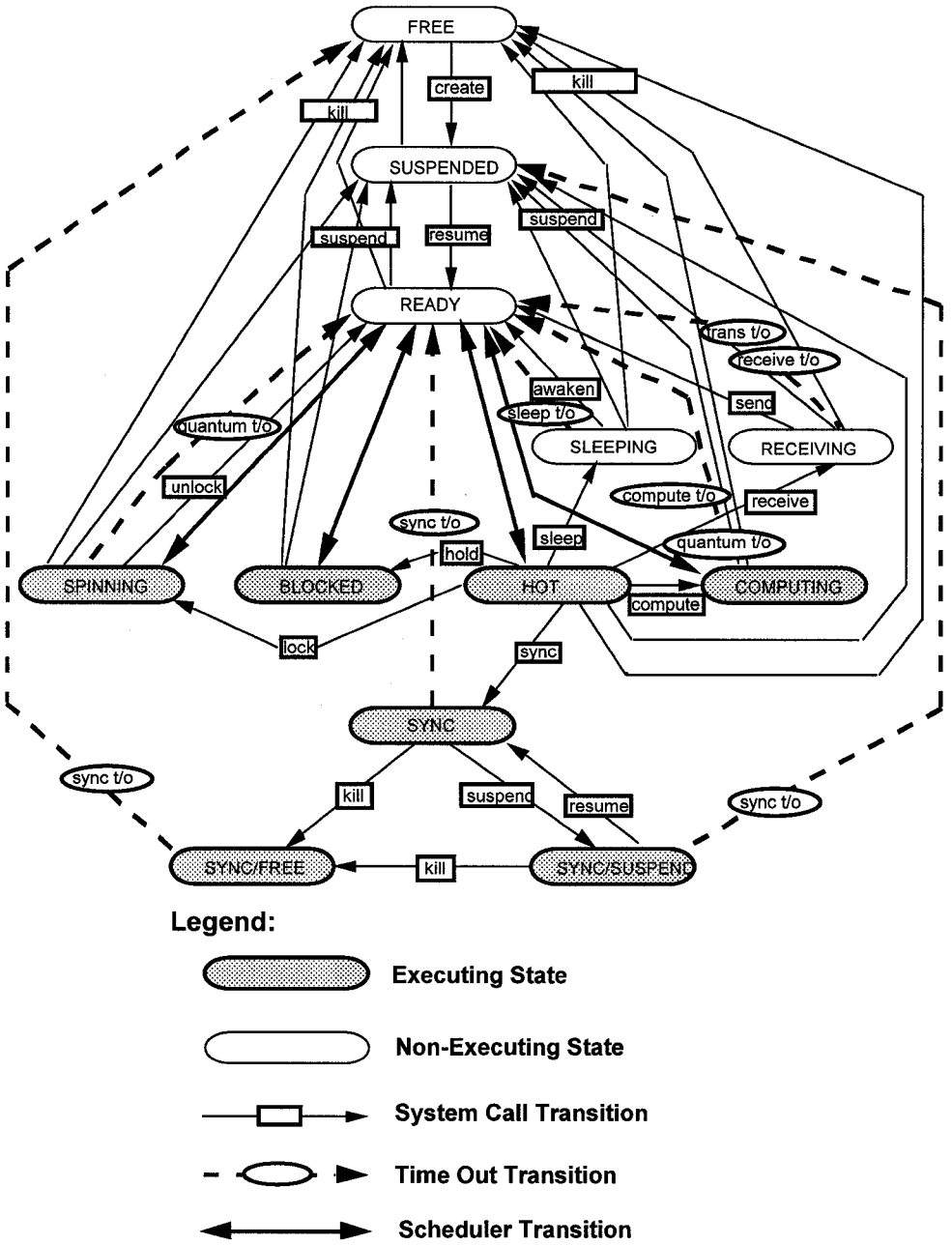


Figure 2.8 PARASOL task states and transitions [21]

Inter-task communication in PARASOL is performed by using ports which are designed for receiving messages. When a message is sent to a task, it is actually sent to a port of a task. If a task is waiting for a message, it is in the receiving state. The receiving state is a non-executing state in PARASOL, so the task is blocked. Ports can be owned by a single task or several tasks, called private ports or shared ports respectively. The messages sent to a port can be queued in FIFO, LIFO or randomly (RAND).

2.4.2.1 PARASOL Internals

In order to demonstrate how PARASOL works, attention should be paid to the PARASOL simulation driver, the global event list, and event handlers.

The simulation is controlled by the simulation driver, which can be reviewed as just another task under PARASOL [22]. The driver is responsible for the initialization of the simulation environment and shutting simulation down after it finishes. The driver also maintains the global event list which stores the future events and advances the simulation time. In PARASOL, the global event list is called Calendar. The current event is stored at the head of the event list. The type of the current event determines the event handler which is called. Once all events at the current time stamp have been processed, the driver will advance the simulation time and jump to the time of the next event.

In PARASOL, there are 14 predefined events and corresponding event handlers. In the event handlers, the state of the task will be changed and other tasks may be scheduled or descheduled. A task is scheduled or descheduled through adding events or removing events. For example, when an End_Quantum event occurs, the End_Quantum event

handler is invoked. In the event handler, the scheduler will always try to find another ready task to execute first, if it is not found, the current HOT task will be rescheduled and executed for another quantum. The End_Quantum event simulates the execution of a task which is only allowed to run a quantum only once. It is for tasks running on a PS processor. PS, Processor Sharing is the scheduling policy used by the analytical model, the simulator runs Round Robin scheduling with a quantum value. The End_Compute event is designed to simulate the general task execution. When a task begins to run, the scheduler adds an End_Compute event into the event list and an End_Quantum event if the task is on a PS processor. Later, when either event happens, the driver comes back to service the task again. During this period, the state of the task is COMPUTING, not HOT, because the driver is doing something else.

Figure 2.9 is used to illustrate the operations of PARASOL. The processing node, Node1 is a 2-processor node using the Processor Sharing scheduling discipline. Task T1 is running on Host1 and T2 is on Host2. The following list corresponds to the labels in the figure.

1. Starting from the label 1 in Figure 2.9, the driver advances the simulation time to the time of the first event in the global event list, the End_Quantum event of task T1.

2. The End_Quantum event of task T1 occurs and its corresponding event handler is invoked.

3. The End_Quantum event handler calls a scheduler function in order to find another ready task.

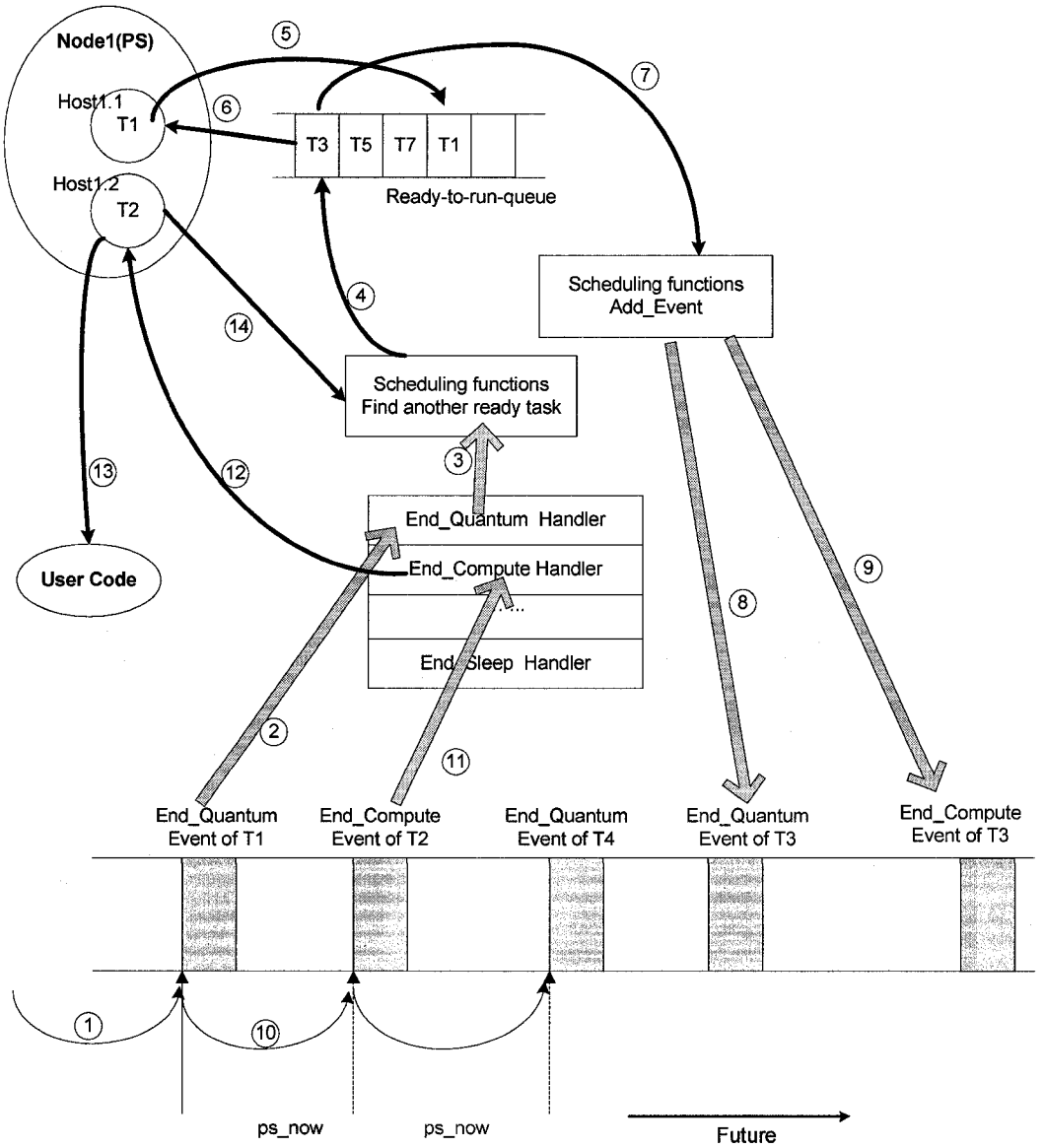


Figure 2.9 PARASOL working process

4. The scheduler function selects the ready task T3 which is the head of the ready-to-run-queue of Node1 in this case.

5. The context of task T1 is saved, T1 is enqueued into the ready-to-run-queue again and the state of T1 becomes READY.

6. Task T3 is dequeued form the ready-to-run-queue, and executed on Host1, then state of task T3 becomes COMPUTING.

7, 8 and 9. The scheduler adds two events for T3, End_Quantum event and End_Compute event, into the global event list.

10. The simulation driver leaves T3, T3 in the COMPUTING state. In the process from label 1 to 9, simulation time stands still. The driver checks the event list, and since there are some events waiting to occur, the driver advances the simulation time to the time of the next event, the End_Compute event of T2. Note that the driver always checks the time first, and prevents any attempt to back up time.

11. The End_Compute handler will be invoked.

12. In End_Compute handler, the state of T2 becomes HOT.

13. The user code resumes execution. The execution of the user code may be complete or may result in another user task become a ready task.

14. After the execution of user code, Host2 becomes a free host. Therefore, the driver calls the scheduling function to find another ready task from the ready-to-run-queue to run on Host2.

In order to run PARASOL, user code must implement a function called ps_genesis( ). ps_genesis( ) is a special reserved function, which acts as the main() function in a C or C++ program. This function provides the start point of the simulator. The configuration of the simulation hardware network is done here. The function ps_genesis( ) calls the

function reaper() to generate two special PARASOL tasks: genesis and reaper. All the user defined tasks are the children of these two tasks as the task hierarchy is used for process control. After task genesis is activated, the simulation is started, and then the simulation driver enters the operation loop and goes through each processor node. The task reaper is used to kill tasks in order to avoid memory leaks.

2.4.2.2 PARASOL built-in scheduler

According to the scheduling discipline, the PARASOL scheduler determines when to perform the switching between tasks, and which task will run next. PARASOL supports some predefined scheduling policies and a user defined scheduling type. PARASOL supports FCFS, PR, PPR, HOL and RAND (random scheduling).

In PARASOL, there is only one ready-to-run-queue in each processing node, and it is implemented by linked lists. In the structure of each node, there is a pointer which points to the task at the head of the ready-to-run-queue. Meanwhile, in the structure of each ready task, there is a pointer which points to the following task of this task in the ready-to-run-queue. There are four scheduling functions in charge of the following situations:

1. When a task becomes a ready task from a non-executing state, the scheduler lets it run immediately if there is a free host or if it has a higher priority than the running task (in PPR scheduling), otherwise the task is put onto the ready queue because no free host is found.

2. When the execution of a task is finished, the scheduler will find another ready task to execute.

3. When the quantum has expired, the scheduler will execute another ready task if it found one or run the current task again if no other ready task was found.

4. When preemption occurs, the task with a higher priority is executed and the lower priority task becomes a ready task again.

The PARASOL scheduler performs the operations of scheduling by generating a future event and adding it into the global event list, and removes an event from the global event list to deschedule a task.

2.4.3 Lqsim Simulator

The Lqsim simulator is a tool designed for simulating layered queueing networks using the PARASOL [22] simulation system.

Lqsim first reads its input model file in one of the two model formats described earlier. Next, Lqsim checks the validity of the input model and initializes the parameters of the simulation. Then, Lqsim invokes the ps_genesis( ) function to start PARASOL execution. During the process of simulation, Lqsim gathers statistical information, outputs the result of simulation, and shuts down the simulator.

In Lqsim, the LQN tasks are classified into ten task types. They are: Client, Server, Multi-server, Inifinite-server, Synchronization-server, Semaphore, Open-arrival-source, Worker, Thread and Signal. Each task type has a different service routine. For this thesis, only the first three tasks and worker tasks are considered.

2.4.3.1 Reference task

A reference task in a LQN becomes a client task in the simulator and performs the client routine. In the client routine, the task loops forever. In each loop, the client task sleeps for a random period of time, then wakes up and sends a request to a server and blocks waiting for the reply.

2.4.3.2 Server task

A server task performs the normal server routine where it loops forever waiting for requests. Its corresponding PARASOL task enters the receiving state and is blocked while the task is waiting for requests. Once a request comes, this server task processes the request through the entry and activity (if there is any), and the PARASOL task becomes a ready task. The service demand is consumed by the parasol task on a host in one or more time slices. After the request is processed, a message is sent back to the sender at a later time. The server task handles another request or waits for other requests to arrive.

Multi- and infinite server tasks are handled differently from single server tasks, shown in Figure 2.10. Each copy of the multi-server corresponds to a worker task which accepts requests from a dispatcher. The dispatcher task serves as the queue for the multi-server and runs on its own private node so that it is scheduled separately from its workers. Clients of the multi-server make requests to the task which will invoke a worker, if available.

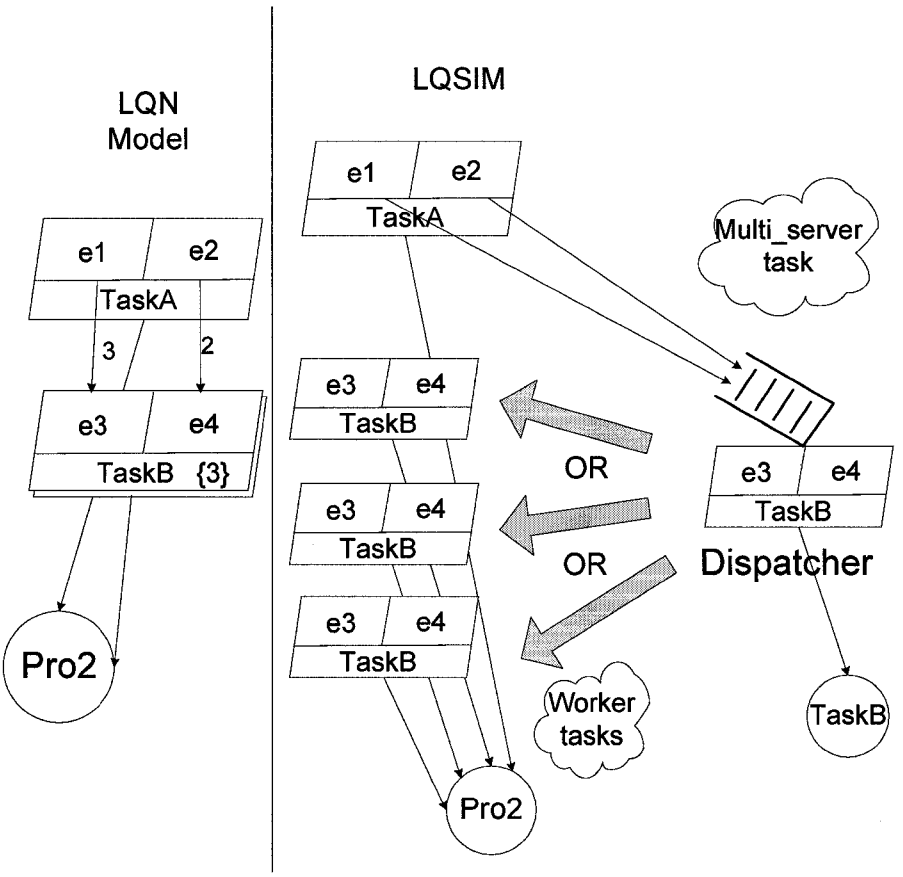


Figure 2.10 A Lqsim Multi-server task

Infinite servers are like multi-servers except that worker tasks are created as needed without limit, so that requests never queue at the dispatcher.

2.5 Summary

This chapter presents the necessary background information for the thesis, which includes the concepts related to fair share scheduling and CFS scheduling, the components of the LQN model, and, the simulation system of the LQN tools: its simulation engine PARASOL and the simulator Lqsim. The next chapter will focus on integrating CFS scheduling into the LQN model and extending the LQN tools to support CFS scheduling.

Chapter 3 CFS scheduling in LQN

This chapter describes how the CFS scheduling policy is integrated into the LQN model and the PARASOL built-in scheduler. The first section describes the algorithm of the CFS scheduler. The second section presents the new element 'group' and the extensions made to the LQN model. The third section describes the design details of the PARASOL CFS scheduler.

3.1 CFS scheduling algorithm

The fair share scheduler described in this thesis is based on the CFS scheduler from the Linux kernel. The key idea of CFS scheduling is to try to reach the goal of 'ideal hardware' with several tasks which are running on a processor in parallel. To achieve this goal, the scheduler will let each task run with a very short time slice and switch among them frequently. The scheduler will choose the task most in need of CPU time to run next.

The CFS scheduler will not result in starvation. A task or a group with a non-zero share will be executed eventually.

3.1.1 The Basic CFS scheduler

In PARASOL, tasks are the basic scheduling unit. Therefore, the basic CFS scheduler

schedules on the granularity of a task, and the basic CFS scheduler is fair to the tasks. This means every task running on a processor will be entitled to the same amount of CPU time.

Suppose that the number of tasks running on a processor is n. In a given time period, T, only one task is running while the others are waiting. The time this task runs is T, and the time each task is entitled to is $\frac{T}{n}$. So this task overruns, and the time of overrunning is $T - \frac{T}{n}$. The need of this task for a processor is decreased. Conversely, a waiting task waits more than it should. The time of the over-waiting is the same amount as it should run in this period, that is $\frac{T}{n}$. So, the need of the waiting task for a processor is increased. 'Fair' is used to express the need for a processor. The greater the value of fair, the graver is the need.

The fair of a task that has become ready is: $\text{fair}_{\text{task}} = 0$.

For a running task, the fair is decreased, $\text{fair}_{\text{task}} = \text{fair}_{\text{task}} - (T - \frac{T}{n})$. (1)

For a waiting task, the fair is increased, $\text{fair}_{\text{task}} = \text{fair}_{\text{task}} + \frac{T}{n}$. (2)

The fair can be recognized as the priority of a task. The fairs of tasks are updated dynamically by the scheduler. The scheduler determines which task should run next by comparing the fairs of ready tasks and the running task. This scheduling mechanism is fair to tasks and similar to Round Robin scheduling to some degree.

3.1.2 The CFS group scheduler

The discussion above described a scheduler based on tasks; this is not satisfying for the goal of a fair share scheduler. A fair share scheduler should be able to be fair to users and groups. This requirement can be realized by the CFS group scheduler. The key point of the CFS group scheduler is that each group is entitled a group share, and each task shares the group share evenly within its group. CFS group scheduling is performed at the processor (in the LQN model) or processing node (in PARASOL level). In PARASOL, the scheduling policy of each processing node may be different.

Consider an example with two groups, Group A and Group B, which are running tasks on the same processor. Group A has 10 running tasks and is entitled to a 40% share, while Group B has 20 tasks running with a 60% share. The CFS Group scheduler enables fairness to Group A and Group B according to their share, instead of being fair to all 30 tasks running on this processor. Since Group A can get 40% of the processor time, the 10 tasks in its group will get a 4% share each. Similarly, Group B runs its 20 tasks using its 60% processor time, so each task in Group B will get a 3% share.

The calculation of task's fair is same as the basic CFS scheduler. The calculation for a group is described as following:

The fair of a group initially is: $fair_{group} = 0$.

For a group with a running task, the time of the group entitled to run is: $(\frac{T}{m}) \times$ share,

where m is the number of groups on the processor.

Therefore, $fair_{group} = fair_{group} - (T - (\frac{T}{m}) \times share)$. (3)

For a group with waiting tasks only, the time of over-waiting is: $(\frac{T}{m}) \times share$.

So, $fair_{group} = fair_{group} + (\frac{T}{m}) \times share$. (4)

The scheduler updates fairs of groups dynamically.

3.1.3 Mixed CFS scheduler

In fair share scheduling, a share can be either a cap or a guarantee. The concept and difference were described in Chapter 2. Most fair share schedulers support one type of share in a system only.

The CFS group scheduler discussed above is the case of the guarantee. The utilization of a group can be greater than its share. If there is no ready task in one group regardless of a short or a long time period, the other groups have a chance to get more CPU time than their entitlement. If the shortage of ready tasks only happens in a short period, the newly ready tasks in this group will boast a greater fair, and get the processor immediately.

A cap is an upper limit of utilization. Using a cap can strictly control the allocation of processor time to a group. The mixed CFS scheduler described here applies the cap concept to the CFS group scheduler. The calculation of the fair of a task or a group is the same as described above. The unique part for a group with a cap constraint is performing some additional checks. If the fair of this group is negative, the group runs out its share temporarily, and all tasks in the group can not be executed for a certain amount of time.

In order to postpone the execution of the group, and considering the efficiency of the computation as well, the fair of the group is increased by two times of the original value of the fair. The detailed algorithm is:

First, increase the fair of the group; $fair_{group} = - fair_{group}$. (If $fair_{group} < 0$) (5)

Then, the scheduler forces all tasks in this group to go to sleep. The sleeping time is:

$T_{sleeping} = 2 \times fair_{group} \times share.$ (6)

3.1.4 Normalize the shares

In the PARASOL and Lqsim, group shares are confined to the range from 0 to 1, and group shares must not be zero. If the sum of group shares for a processor is greater than 1, it results in a conflict to the principle of CFS group scheduling. If the group shares are guarantees, a group should get a processor resource at least equal to its share. It is impossible for the scheduler to allocate more than 100 % of the resources to groups; therefore, it is necessary to normalize the shares to one. If one or more groups have cap shares rather than guarantee shares in the model, the situation is more complex. There is a tradeoff to reduce cap shares or guarantee shares or both. The design decision for this work was to keep the relative ratios among the groups, thus reducing all shares regardless of whether they are guarantees or caps.

3.2 Integration CFS into the LQN model

The PARASOL CFS scheduler integrates the three features from the previous section, namely the basic CFS scheduler, the CFS group scheduler and the mixed CFS scheduler.

These three features are incorporated into the LQN model to realize the goal of tuning the performance of computer systems by the CFS scheduler.

3.2.1 Extension to LQN model

In order to be compatible with the CFS group scheduling, a new element called 'group' is added to the LQN model. The group element is designed as an optional element, so that models which do not need group scheduling require no change. Figure 3.1 shows the changes made to the LQN Meta model. The scope of the group element is between processor and task. A group contains one or more tasks. If no group is defined for a processor, one or more tasks should be defined under the processor directly. Therefore, a processor can support both groups and non-group tasks. However, group tasks and non-group tasks can not exist on one processor. Extensions to the LQN component model, the LQN input model, and the graphical notation are required to support the group concept.

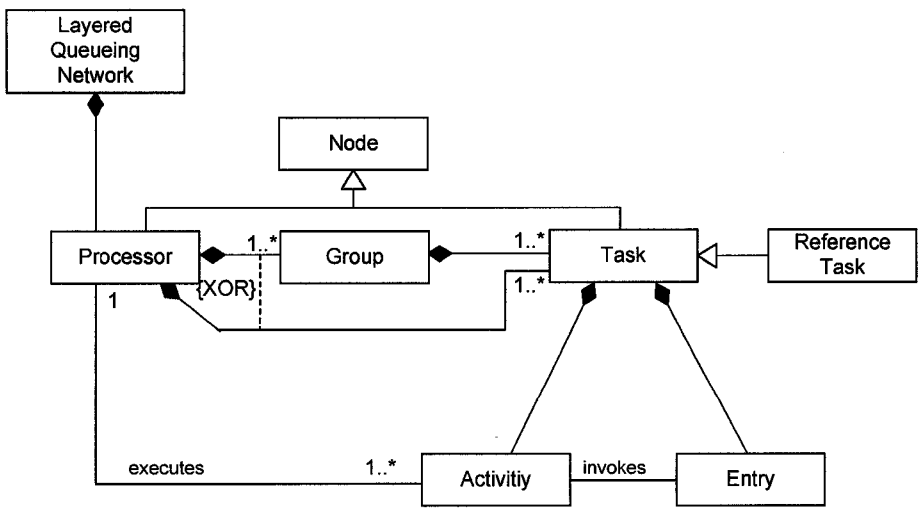
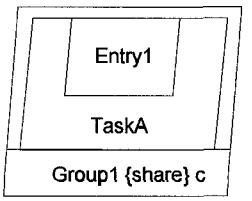
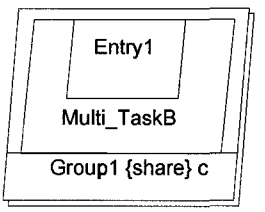
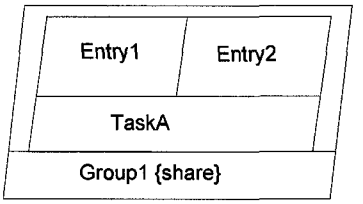
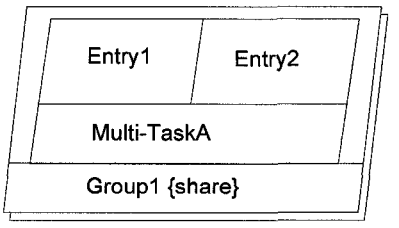


Figure 3.1 New LQN Meta model (simplified)

Table 3.1 shows the new notations for the graphical notation. The new notation for the group element adds another parallelogram outside the normal task notation, which indicates the task is contained in a group. The group name and its share are specified at the bottom of the outer parallelogram. The optional letter 'c' after the braces of the share is the notation for the cap.

Table 3.1 The graphical notations of 'group' element

Task type	Element 'group' graphical notations
A normal task with one entry is contained by a group. The share of the group is a cap.	
A multi-server task with one entry is contained by a group. The share of the group is a cap.	
A normal task with two entries. The share of the group is a guarantee.	
A multi-server task with two entries contained by a group. The share of the group is a guarantee.	

Consider the simple input model shown in Figure 3.2, which illustrates the usage of new notation. TaskA and TaskB are in the same group, GroupA, with a share of 0.75, while

TaskC is in the GroupB with a share of 0.25. TaskB is a multi-server task. No cap is defined in this input model.

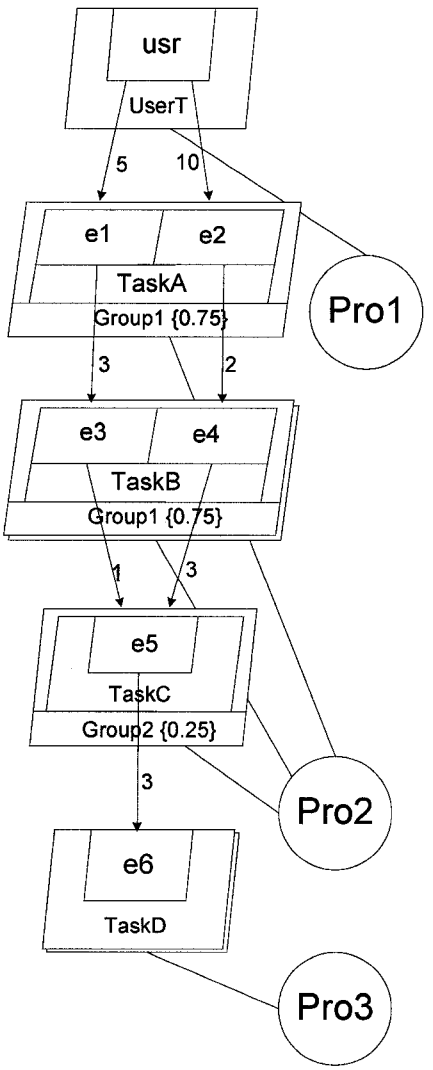


Figure3.2 The LQN input model with group definition.

3.2.2 Extension to LQN Core schema

In the LQN core model, the newly added group element is defined under the processor element. It is an optional element which is realized by a choice element to implement its

optional property. Figure 3.3 shows the new LQN Core schema graphically and the XML schema definition can be found in Appendix A.

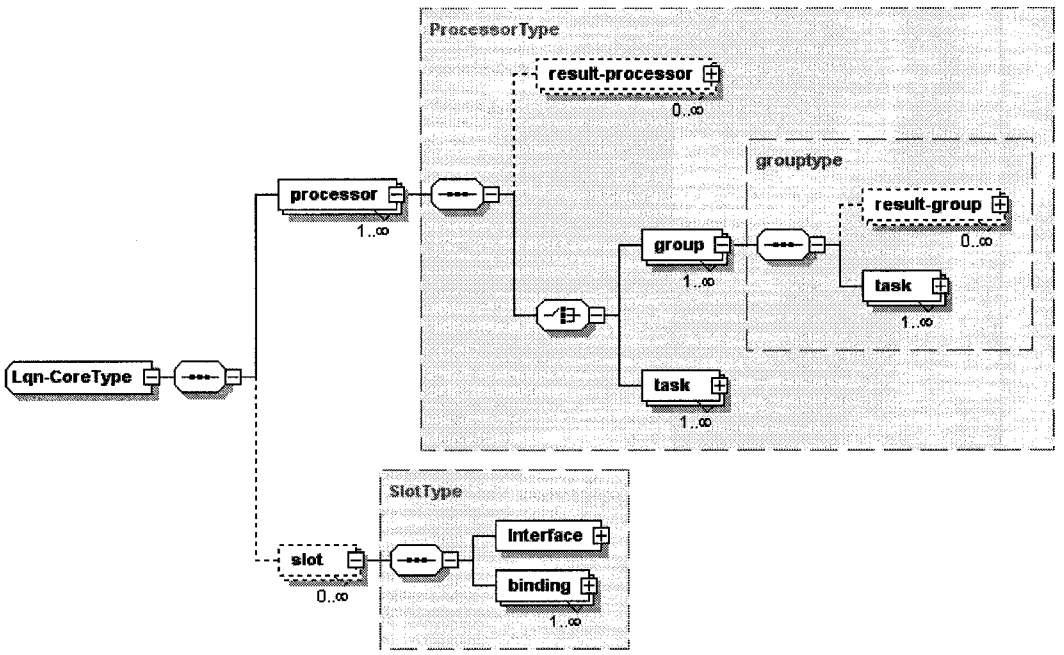


Figure 3.3 New LQN Core schema

The group element is specified by a complex type element, GroupType. GroupType contains an optional result-group element and task elements. Element result-group is of type OutputResultType to store the statistics information of a group element. The attributes of GroupType include name, share and cap_flag. A group element must contain one or more task elements.

In the XML Format Input Model, a new group tag is added to define group elements. A task is not grouped into a group if there is no group tag between the processor tag and task tags. Listing 3.1 shows the simplified XML format input file of the example in the previous section.

Listing 3.1 An XML format LQN input model

```
<?xml version="1.0" ?>
 <lqn-model name="groupmodel"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="/usr/local/share/lqns/lqn.xsd">
<solver-params comment="This is a example" conv_val="1e-06"
  it_limit="100" print_int="1" underrelax_coeff="0.9" />
   <processor name="Pro1" scheduling="fcfs">
      <task name="UserT" multiplicity="10" scheduling="fcfs">
      ...
      </task>
    </processor>
   <processor name="Pro2" scheduling="cfs" quantum="0.1" >
      <group name="Group1" share="0.75" cap_flag="0" >
         <task name="TaskA" multiplicity="20" scheduling="cfs">
            ...
         </task>
         <task name="TaskB" multiplicity="20" scheduling="cfs">
            ...
         </task>
        </group>
        <group name="Group2" share="0.25" cap_flag="0" >
           <task name="TaskC" scheduling="cfs" >
           ...
           </task>
        </group>
       </processor>
  </lqn-model>
```

In the implementation, there are two functions for reading tasks from a XML format input file, one reads tasks directly under a processor tag, and the other reads tasks under a group tag. The parser will recognize the group definition by comparing the input file with the new LQN core model. In the LQN tool set, lqn2xml can convert an old style LQN input file to an XML input file.

3.2.3 Extension to the Old Style LQN Input Model

This section describes the changes to the old style input file syntax. In the old style LQN input file format, a new section is added to describe the group information, which consists of the name of a group, group share, cap_flag, and the name of processor which

the group belongs to. Listing 3.2 shows the syntax by the input model in section 3.2.1. Further modifications were made to the definition section of processors and tasks. In the processor definition section, the CFS scheduling type was added. The CFS scheduling flag consists of a letter 'c' and followed by a quantum. In the task definition, a group flag is added to each task within a group. The group flag consists of a letter 'g' and followed by the group name to which the task belongs to. The complete grammar, including the changes describe here, are found in Appendix B.

Listing 3.2 An LQN model in Old style LQN input file.

```
P 0 # processor definition section
p Pro1 f #Processor Pro1 (FIFO)
p Pro2 c 0.1  # c for CFS, and quantum is required
p Pro3 f #Processor Pro3 (FIFO)
-1

U  0 # Group Information section
#g   Group_name  Group_share  [cap_flag ]  Processor_name
g Group1 0.75  Pro2
g Group2 0.25   Pro2
-1

T 0 # Task Information section
#SYNTAX: t TaskName RefFlag EntryList -1 ProcessorName
#[multiplicity] [group flag]
 # group flag =g group name
 t UserT r user -1 Pro1 m 10
 t TaskA n e1 e2 -1 Pro2 g Group1
 t TaskB n e3 e4 -1 Pro2  m 20 g Group1
 t TaskC n e5 -1 Pro2  g Group2
 t TaskD n e6 -1 Pro3 m 10
 -1
```

Here, the scheduling type of the processing node, 'Pro2' is CFS, with the maximum quantum of 0.1 time unit. The quantum is described in a later section. The group notation does not affect the definition of entries and activities in the LQN input model.

3.3 Parasol CFS scheduler

In this section, the implementation details of the CFS scheduler are described.

Figure 3.4 shows the changes to the PARASOL class diagram in Figure 2.7. There are two new classes, Group and CFS-rq. The XOR operator indicates a processing node can not have group tasks and non-group tasks at the same time. The CFS-rq is a ready-to-run-queue in a CFS processing node, and it will be discussed in detail in the following section.

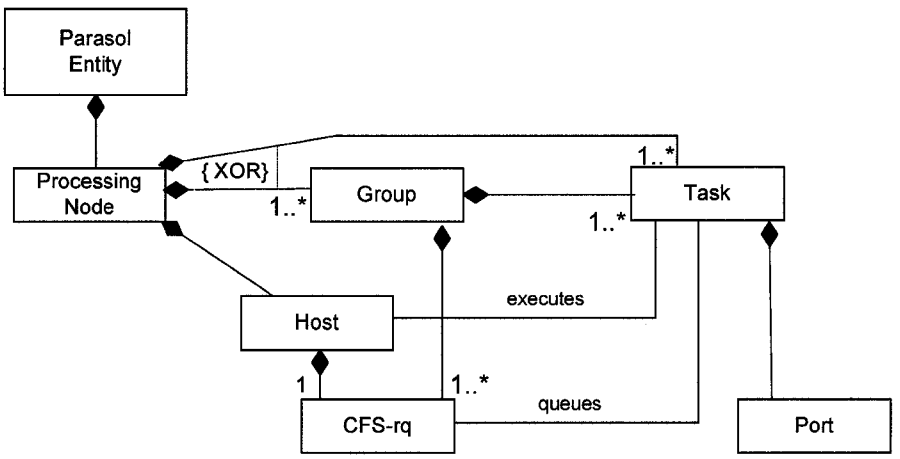


Figure 3.4 New PARASOL class diagram

3.3.1 Parasol CFS ready-to-run-queue

In the original PARASOL kernel, each processing node has a single ready-to-run-queue, shown in Figure 3.5. All tasks running on the node stay in this ready-to-run-queue when they are in the READY state. In order to support CFS scheduling in PARASOL, the existing ready-to-run-queue is replaced with a red-black tree sorted by priority. For the CFS scheduler, the ready-to-run-queue is ordered by fair. This ready-to-run-queue is

called a CFS-rq. Note that unlike the original PARASOL ready-to-run-queue, the CFS-rq is not node-wide, rather one queue exists for each host on a node shown in Figure 3.6.

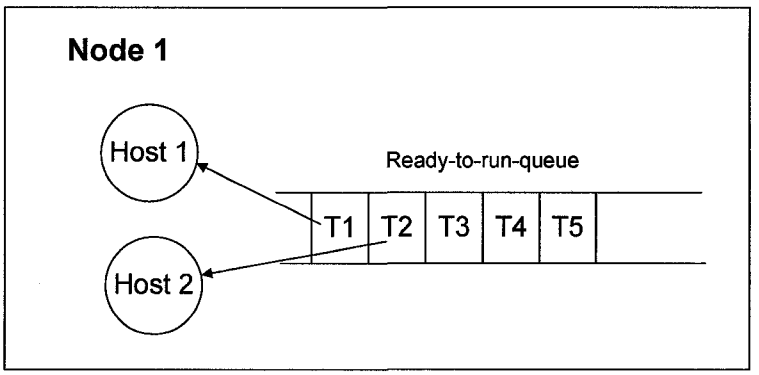


Figure 3.5 The structure of an original ready-to-run-queue.

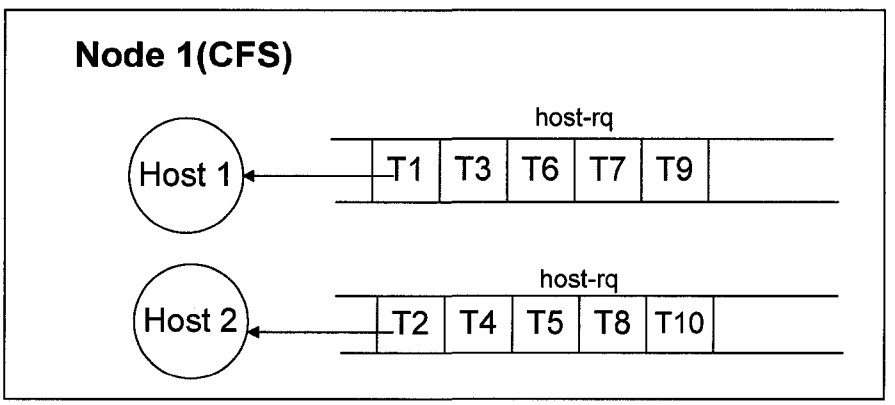


Figure 3.6 The structure of the host-rqs

3.3.1.1 Host-rqs and group-rqs

In the CFS scheduler, CFS ready-to-run-queues are organized in a hierarchical way. Each host has a CFS-rq, called the "upper-rq" or "host-rq". If there is no group definition in the node, the ready tasks stay in the host-rqs and wait to be executed, and the CFS scheduler performs the basic CFS scheduling using host-rqs only. The nodes in a host-rq point to ready tasks directly and the first task in each host-rq will be executed next. Each

host-rq can only have one running task, and the running task is not dequeued from the host-rq when it is running on a CPU.

If there are task groups, under a specific host-rq, each group owns a CFS-rq as well. These CFS-rqs are called "lower-rqs" or "group-rqs". In this situation, the entities in a host-rq point to their corresponding group-rqs instead. The ready tasks are enqueued onto their own group-rqs. Figure 3.7 shows an example where three group-rqs are nested in each host-rq. There are pointers from the nodes in a host-rq to each group-rq. Therefore, under a host-rq, only one group-rq can have a running task. The first ready task in the first group-rq will be executed next. In this implementation, there are two levels of CFS-rqs only.

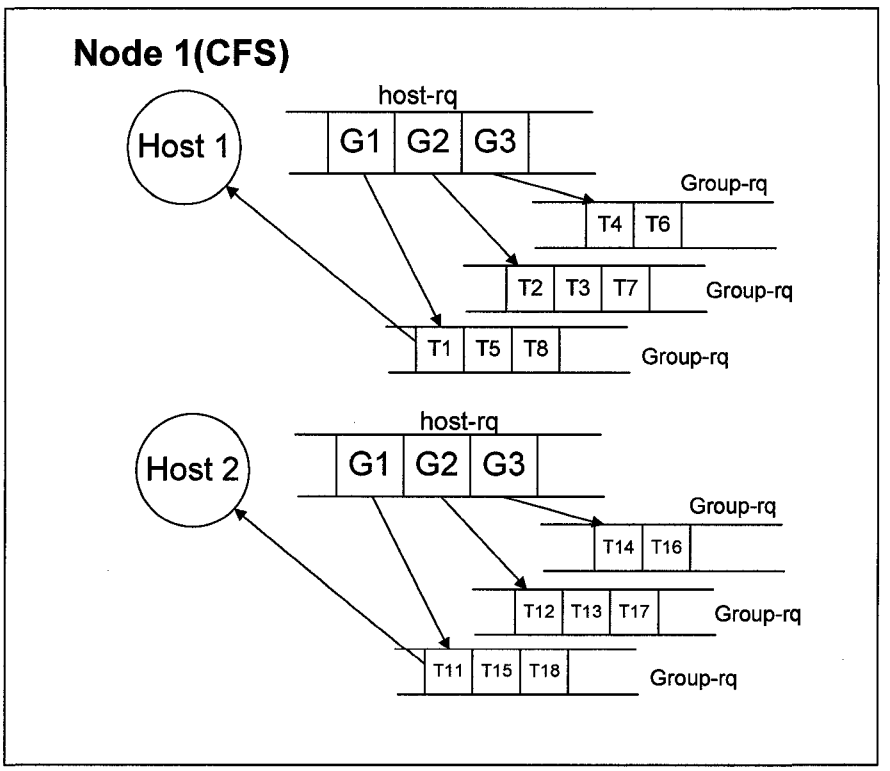


Figure 3.7 The hierarchical structure of group-rq and host-rq

According to the CFS scheduling algorithm described earlier, every execution of a task alters the priorities of all tasks and groups. In the implementation, the key point is how to update the priorities of tasks and groups in CFS-rqs efficiently.

3.3.1.2 Structure of a CFS-rq and nodes in a CFS-rq

In PARASOL, a CFS-rq is implemented by a red-black tree structure, because it guarantees the worst-case execution time in logarithmic time for its most operations, such as insertion, deletion, and search. The red-black tree data structure is very efficient in practice, for example there are a number of occurances in Linux kernel, such as schedulers, the CD\DVD driver and the Virtual memory areas (VMAs) [41].

The struct 'rb_node' represents the nodes in a red-black tree, and the struct 'ps_cfs_rq_t' represents a CFS-rq.

```
struct  rb_node {       /* struct for a red-black tree node */
            int       color;
            struct rb_node *rb_left;     /* left child node;  */
            struct rb_node *rb_right;    /* right child node; */
            struct rb_node *rb_parent;   /* Parent node;      */
            double              key;
            int       entity;
}rb_node;
```

In an rb_node, the color can be either RB_RED or RB_BLACK. All leaves are sentinel nodes, and are represented by NIL. The entity can be the index of a task or a group.

When a host-rq consists of several group-rqs, the entity of an rb_node in the host-rq stands for a group index.

```
struct  ps_cfs_rq_t {                          /* struct for red-black tree */
        int       node;                        /* node location          */
        int owned_by ;                         /*owned_by a group        */
        int nready;                            /* number of ready to run tasks */
        struct rb_node * root;                 /* root node              */
        struct rb_node * leftmost;             /* the node with smallest key      */
        struct rb_node * rbnode;               /* a pointer to the node in host-rq  */
        struct rb_node * current;              /* the node for the running task  */
                                               /*  or a group with the running task */
        double delta_fair;          /* tract the change of fair of the ready tasks */
        double sched_time;                     /* Task schedule time      */
} ps_cfs_rq_t;
```

In a CFS-rq, the pointer stored in the leftmost node points to a task or a group which will run next. The host-rqs are built during the construction of a CFS processing node, while the group-rqs are built during the instantiation of groups.

3.3.1.3 Operations of a CFS-rq

All the enqueueing and dequeueing operations of CFS-rqs are done by the red-black tree insertion and removal operations. Tree operations that form CFS-rq operations are separated in order to maintain and reuse the red-black tree in the future.

In order to maintain a red-black tree, some basic tree operation functions are needed. They include the initialization of rb_nodes and ps_cfs_rq_ts, insertion, deletion, and searching of an rb_node. After insertion or deletion, the properties of red-black tree must be maintained, several left or right rotations have to be applied to keep the balance of the

red-black tree. The detailed algorithm can be found in [31]. The main operations on the CFS-rqs are described below.

1. Enqueue a task

When a task becomes a ready task, the task is enqueued onto a CFS-rq. The pseudo code is:

```
void enqueue_cfs_task( ps_task_t *task)
{
        ps_cfs_rq_t  * host_rq = task->hp-> host_rq;
        if (the task is not contained by a group)
        {  Update the running task of the host_rq ;
                update the host-rq;
                Insert_rb_node (host-rq, task->key);
        }else{
                ps_cfs_rq_t  *   group_rq = task-> rq;
                Update the running task of the group_rq;
                update the group-rq;
                Insert_rb_node (group-rq, task->fair);

                if (the group-rq is not on rq){
                        update the host-rq;
                        /*  if the group run out its cap share          */
                        if (the time stamp > ps_now)
                                adjust the fair of the group-rq;
                        /* enqueue the group_rq onto the host_rq    */
                        ps_group_t * group = ps_group_ptr(group_rq -> owned_by)
                        insert_rb_node (host-rq, group->fair)
                }
        }
}
```

Note that once a task is enqueued, it stays in the CFS-rq while it is running or waiting until it goes to some other non-executing states.

If the share of a group is a cap and the group runs out its share, the rb_node pointed to the group-rq will be dequeued from the host-rq and the time stamp of the group is assigned a future time. When a new ready task arrives, its group-rq has to be enqueued onto the host-rq again. In this situation, adjusting the fair and the time stamp of the group-rq is compulsory. After the adjustment, the new ready task may end up going to sleep shortly.

2. Dequeue a task

If a task is no longer a ready task or a running task, the task will be dequeued from the CFS-rq. The pseudo code is:

```
void dequeue_cfs_task( ps_task_t *task)
{
        if (the task is not on the rq)
                ps_abort ("A ready task is missing.");
        delete_rb_node (task->rq, task->rb_node);

        /* the task->rq is a group-rq AND no ready task is in this group-rq */
        if (task->rq ->owned_by != -1){
                ps_cfs_rq_t * group_rq =task->rq;
                if (group_rq -> nready ==0){
                        ps_cfs_rq_t * host_rq =task->hp ->host_rq;
                        delete_rb_node (host-rq, task->rq->rbnode);
                }
        }
}
```

3. Find the next task to run.

The task next to run is the task pointed by either the leftmost node of a host-rq or the leftmost node of the leftmost group-rq. The pseudo code is:

```
int  find_cfs_run_task( ps_cfs_rq_t *host-rq)
{
        struct rb_node     aRb_Node;
        aRb_Node = host-rq -> leftmost;
        if (aRb_Node ==NIL)
                return NULL_TASK;
        if (aRb_Node stands for  a group-rq) {
                ps_cfs_rq  *group_rq = get_group_rq(aRb_Node);
                aRb_Node = group_ rq -> leftmost;
        }
        return aRb_Node ->entity;
}
```

4. Find a task next to a specified task

Find a successor task to this specific task. The pseudo code is:

```
int  find_next_cfs_task( ps_task_t *task)
{
        struct rb_node     aRb_Node;
        /* find the successor node of the task is the same rq; */
        aRbNode = succesor( task->rq, task->rb_node);
        if (aRb_Node !=NIL)
                return aRb_Node ->entity;
        else if (task->rq ->owned_by != -1)        /* task->rq is a group-rq*/
        {       /* find the successor of the group-rq */
                ps_cfs_rq_t * group_rq =task->rq;
                ps_cfs_rq_t * host_rq =task->hp ->host_rq;
                aRb_Node = successor (host-rq, group_rq ->rb_node);
                if (aRb_Node !=NIL){
                /* return the corresponding task of the leftmost node of this group_rq*/
                        ps_cfs_rq  *group_rq = get_group_rq(aRb_Node);
                        return group_ rq -> leftmost-> entity;
                }
        }
        return NULL_TASK;
}
```

3.3.2 Parasol Built-in CFS scheduler

The first job of the CFS scheduler is updating the priorities of the running and waiting tasks, and group-rqs (if any), updating CFS-rqs, and maintaining the correct positions of the rb_nodes in a CFS-rq. The second job is scheduling tasks, which means choosing a task eligible to run next according to the fair of tasks. The CFS scheduler tries to run each ready task for a very small amount of time (called the quantum), and creates the illusion of concurrency. Since task preemption by a higher priority task is not allowed in the CFS scheduler, the reschedule happens after a task has finished running or its quantum has expired. The quantum is discussed in detail in next section.

As a result of the CFS algorithm, the priorities of all the tasks in a CFS-rq will be changed after every execution of any task in the same CFS-rq. Meanwhile, if group-rqs exist, the changes will propagate to the other group-rqs through the host-rq.

3.3.2.1 Update priority

Updating priority includes updating the priority of the running task, the other ready tasks, and group-rqs (if there are any). Updating the priority of a task means that its position in its CFS-rq needs to be updated as well. Updating the position of a task in a CFS-rq can be done by dequeueing this task first and then enqueueing the task again. Although the red-black tree guarantees the worst case execution time, the execution time of tree node operations is still significant. Therefore, updating each node in the tree frequently must be avoided.

Since updating the running task priority is inevitable after each execution, the ready tasks should not be updated every time. To accomplish this, the fixed relative positions of the ready tasks in a CFS-rq must be maintained, i.e., the key of the rb_node will keep still as time elapses after the task is enqueued onto the CFS-rq. Therefore, the determining the value of the key of a rb_node is very tricky, and is

key = the delta_fair of the CFS-rq –fair of the task. [40]

The delta_fair of a CFS-rq tracks the change of the fair of ready tasks. When a ready task is waiting for a processor, its fair increases, and the delta_fair of its CFS-rq does the same. If the changes of these two variables have the same value, then the key for its rb_node does not change until it runs on a CPU. Therefore, once a task is inserted into a CFS-rq, its rb_node will keep its relative location until the rb_node becomes to the leftmost node and the task is executed eventually. As a result, the updating rb_nodes of ready tasks will not happen frequently. This will reduce the computing cost.

3.3.2.1.1 Update a ready task

Based upon the discussion above, updating a ready task means increasing its fair with the amount of the change of CFS-rq's delta_fair in its waiting period. This operation happens in two cases. The first case is when the task becomes the leftmost node in its CFS-rq, and before it is executed on the CPU. The other case is before it is dequeued from its CFS-rq. The pseudo code is:

```
void update_cfs_ready_task( ps_task_t * task)
{
        if ( time stamp of task >ps_now)
                ps_abort();
        ps_cfs_rq_t * rq = task->rq;
        if (the time stamp of the rq < ps_now)
        {
                if (the running task is in this rq)
                        update the running task;
                else
                        update the time stamp of rq;
        }
        update the fair of the task;
        update the time stamp of the task;
}
```

3.3.2.1.2 Update the running task

After the execution of a running task, the scheduler will lower the priority of the running task by decreasing its fair value using Formula (1) and (2) in the Section 3.1. Simultaneously, it will update the delta_fair and the time stamp of the task's rb-node in its CFS-rq. This operation can be reached through the End_Quantum event and the End_Compute event in the PARASOL scheduler. Some other cases also invoke updating the running task, such as enqueueing and dequeing a task. The pseudo code is:

```
void update_cfs_run_task( ps_task_t * task)
{
        update the fair of the task;
        ps_cfs_rq_t * rq =task->rq;
        update the delta_fair of the rq;
        /*adjust the position of the rb_node of the task; */
                dequeue_cfs_task (task);
                enqueue_cfs_task (task);
        update the time stamp of the task and the rq;
}
```

3.3.2.1.3 Update group-rqs

If the running task is contained in a group-rq, the scheduler needs to lower the priority of this group-rq, and then increases the priority of the other group-rqs which belong to the same host-rq. The method is similar to updating the running task, except that all group-rqs of waiting groups need to be updated individually. Because each group has a different share, the fair of each waiting group does not increase with the same value. Therefore, the key of the corresponding rb_node for each group-rq can not remain unchanged. The pseudo code for updating group-rqs is:

```
void update_group_rq( ps_cfs_rq_t * host_rq)
{
        ps_cfs_rq_t * current_group_rq= get_group_rq(host_rq->current);
        update the priority of the current_group_rq;
        update the delta_fair of the host-rq;
        adjust the position of  the current_group_rq;
        update the time stamp of the current_group_rq;
        rb_node * rbnode =host_rq->leftmost;
        for each rbnode in the host_rq  do:{
                ps_cfs_rq_t * group_rq= get_group_rq(rbnode);
                if (the group_rq is the current_group_rq) then
                        continue;
                else{
                        update the priority of the group_rq;
                        adjust the position of the group_rq;
                        update the time stamp of the group_rq;
                }
        }
}
```

This operation will consume quite a lot of computing time, because each group-rq is updated every time. Recall that tasks in different groups do not compete with each other.

Therefore, under a host-rq, the priority for tasks in the group-rqs without the running task will not be changed.

The update to group-rq always follows the update to the running task in order to maintain every node in its correct position.

3.3.2.2 Schedule tasks

In the PARASOL scheduler, a reschedule happens whenever the state of a task is changed. Extensions are made to the scheduler functions to support CFS scheduling. A task is referred as a CFS task if the task is scheduled by the CFS scheduler.

1. When a CFS task becomes a ready task from a non-executing state, the CFS scheduler first looks for a host with the minimum number of tasks in its group or in the host. Then the scheduler enqueues the task onto the host-rq or its group-rq. It may run immediately if this host with minimum load is a free host. The pseudo code is:

```
void find_host_cfs( ps_task_t *task)
{
        int host = find_min_host(task);
        enqueue_cfs_task( task);
        if (the host is a free host) {
                set the task to run;
                add an End_Quantum event;
                add an End_Compute event;
        }
}
```

2. When the execution of a task is finished, the scheduler will choose another ready task to execute. The running task must be dequeued before this function is called. The pseudo code is:

```
void find_ready_cfs( ps_node_t *np, ps_cpu_t *hp)
{
        int task = find_cfs_run_task( hp->host-rq);
        If (task !=Null){
                set the task to run;
                add an End_Quantum event;
                add an End_Compute event;
        }
}
```

3. When the quantum for a task has expired, the scheduler will choose another ready task to run if possible or run the task again if no other eligible task is found. The host-rq and group-rqs must be updated first. The pseudo code is:

```
find_priority_cfs(ps_task_t *task)
{
        update_cfs_run_task(task);
        if (the group of this task has a cap share AND  the fair of the task group <0)
        {       cap_handler (task->rq );
                return;
        }
        find_cfs_task_run( task-> hp ->host_rq);
        newTask = the address of new task;
        if (newTask != Null AND newTask != task)
        if (compare_fair( newTask, task)==True)
        {
                saving the running task;
                remove its End_Compute event ;
                set newTask to run;
                add an End_Quantum event for newTask;
                add an End_Compute event for newTask ;
        }else
                add an End_Quantum event for original running task
}
```

Here, the function "compare_fair( )" is used to compare the priorities between the running task and the ready task in leftmost position in the host-rq. If the function returns False, the ready task will not be scheduled and the running task is executed again. If these two tasks are not in the same group, the function will return True.

In this scheduler function, the function "cap_handler( )" deals with the cap share, and it is invoked only when the fair of the group with a cap share is negative, which mean the group runs out of its share temporarily. The pseudo code is:

```
void cap_handler(ps_cfs_rq_t * group_rq){
        ps_group_t * group= ps_group_ptr(group_rq ->entity);
        /* Change the fair of the group from negative to positive */
        group->fair= -group->fair;
        /* Calculate the sleep time period*/
        double   delta=2* group-> fair  * group ->share;
        For each task in the group-rq, do:
        {
                Dequeue the task;
                Remove the End_compute event;
                Add an End_Sleep event for;
        }
        set the time stamp of group-rq a future time = ps_now+delta;
        dequeue the rb_node of the group-rq;
        find_ready_cfs();
}
```

3.3.3 Quantum

The nature of the CFS scheduler is to execute a task for a very small time period. Therefore, a quantum must be defined for each task to decide how long it can be run each time. With the round robin scheduling policy, a fixed quantum is used. With the CFS

scheduler, an adjustable quantum is used instead. Apparent concurrency increases as the size of the quantum decreases. However, the consequence of using a small quantum is the computing expense will be very high. It is reasonable to choose a quantum associated with the number of ready tasks. Therefore, the quantum of a task is the service demand divided by the number of ready tasks at the time when it becomes a ready task.

Using this heuristic can result in a quantum which is too small or too large, so a minimum and maximum quantum is also defined. The maximum quantum is provided by the input model, the minimum quantum was chosen to be five times smaller than maximum quantum. This range allows the scheduler to choose values for the quantum but prevents excessively small values from being chosen which could inflate the simulation runtime unreasonably. The dynamic quantum is calculated by the execution time the task need divided by the number of ready tasks. If the dynamic quantum is between the range of the minimum quantum and maximum quantum, the task uses the dynamic quantum. Otherwise, the maximum or the minimum quantum is used instead.

3.4 Lqsim simulator

The LQN simulator, Lqsim, is built on the top of the extented PARASOL engine. After the introduction of 'group', some modifications must be made to the tasks in Lqsim. The following is a detailed description.

In general, it is not recommended that a client task use CFS scheduling because there is no need for competing for a processor resource. Therefore, the CFS scheduling discipline is ignored by client tasks and FCFS scheduling is used instead.

For a simple single server task, the server routine in Lqsim does not change. However, when the PARASOL task is instantialized in the PARASOL kernel, it carries the group information of the server task. When the PARASOL task becomes a ready task, it is enqueued onto its group-rq, and competes for CPU resources with the other tasks in the same group.

A multi-server task in the LQN model creates a number of worker tasks on the processor node defined by the model, and a dispatcher task running on its own PARASOL node. If this multi-server task is contained in a group, a new group will be created which contains the worker tasks of the multiserver, and the group will receive the share of the task in the LQN model. Each worker task will evenly receive a portion of the new group's share. The dispatcher task still runs on its own node and is scheduled independently.

Figure 3.8 is used to describe this process. Initially, TaskA and TaskB are in the same group, Group1, with a share of 0.75. If both tasks were single servers, both TaskA and Task B will receive 37.5% of the overall processor share. Since TaskB is a multi-server, a new group, TaskB-group, is created and given a share of 0.375. Each of the worker tasks is given 0.375/3 =0.125. In Lqsim, before the creation of the instances of worker tasks, the new group should be created in both Lqsim and PARASOL.

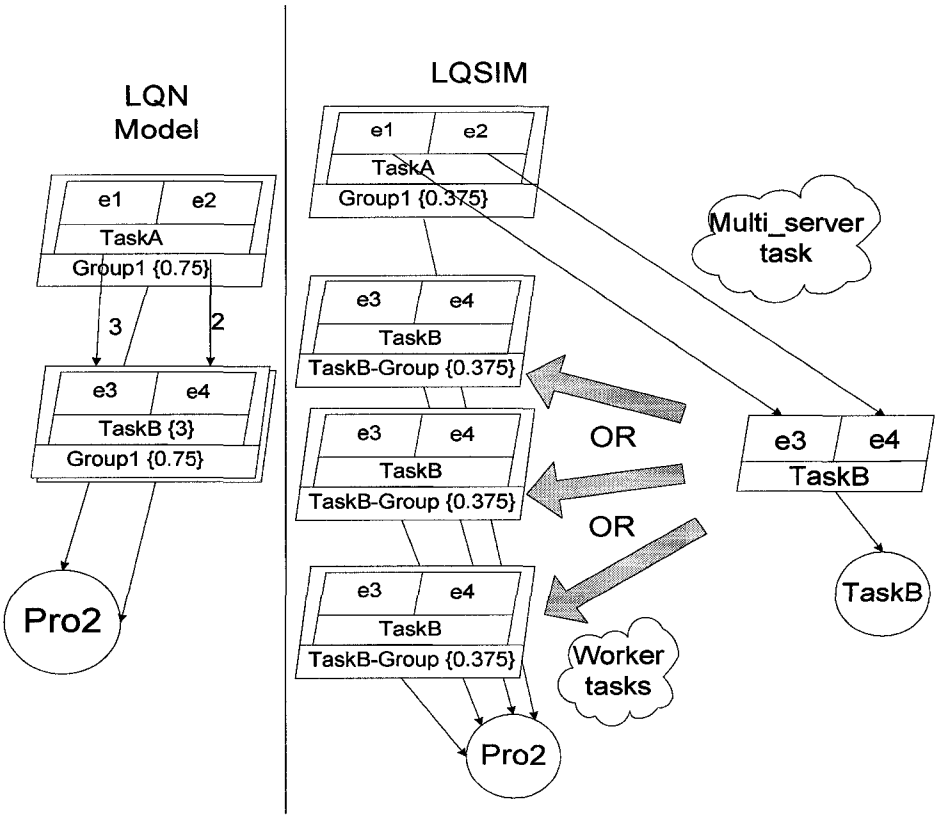


Figure 3.8 A Lqsim multi-task contained by a group

The psuedo_code for the group adjustment is:

```
void group_adjustment (task_class * cp )
{
   group * orign_group =find_group_ptr(cp->group_id);
   if (orign_group->total_task_classes > 1){
         double new_share = orign_group->share / orign_group->total_task_classes;
         /*Add a new group to model with new share and new name;*/
         group * new_group = add_group (cp->name, new_share, cp->proc_id);
         Update group share of orign_group;
         cp-> group_id = new_group ->group_id;

         /* Create a new group in PARASOL */
         build_group(new_group->group_id, new_group->share, new_group->name);
}
}
```

Typically, an infinite server is used to model pure delay. In lqsim, worker tasks are created as needed without bound. In the limit there will be an infinite number of worker with zero share each. Therefore, it is unwise to place this kind of task into a group. If the CFS scheduler is assigned to an infinite server in the input model, Lqsim will run the FCFS scheduler instead. As a result, the group definition will be ignored.

3.5 Summary

The design and implementation of the CFS scheduler were described in the previous sections and the modifications of the Lqsim simulator detailed. The verification should be performed next in order to ensure the CFS scheduler is working correctly.

Chapter 4 Demonstration of the PARASOL CFS scheduler

In this chapter, a very simple LQN input model is input to the Lqsim simulator, to show whether the CFS scheduler is working correctly or not. Further, comparisons with the FCFS and PS schedulers are presented.

The Basic input model is shown in Figure 4.1. There are two user tasks (reference tasks)

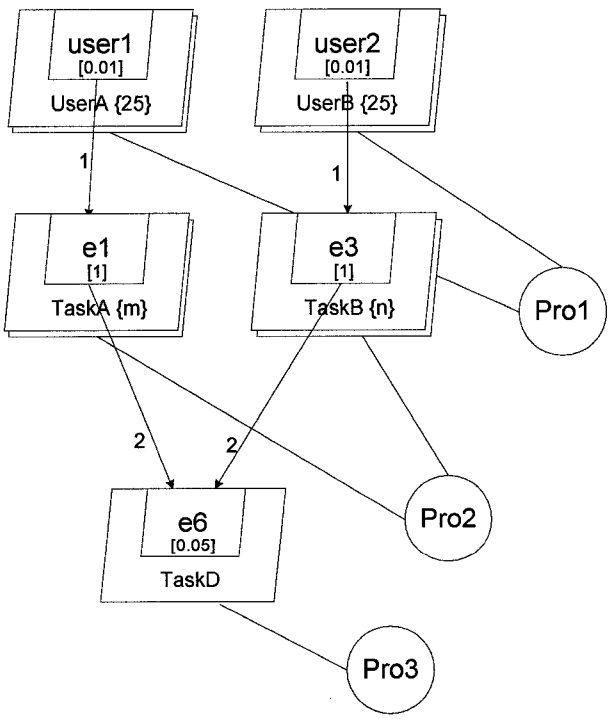


Figure 4.1 Basic Input model for testing

UserA and UserB, both with 0.01 second service demand. They make requests to TaskA and TaskB once respectively. The service demands of TaskA and TaskB are 1 second. TaskA and TaskB call TaskD twice in a server cycle. TaskA and TaskB run on the same node, Pro2. In the testing, the CFS scheduling is applied to Pro2. The Multiplicity of UserA and UserB are 25. In the Figure 4.1, m and n stand for the multiplicity parameterof TaskA and TaskB respectively.

4.1 The basic CFS scheduler

In order to verify the basic CFS scheduler, it will be compared with the PS scheduler using the basic input model. Tables 4.1 and 4.2 show the utilizations and response time of TaskA and TaskB in two cases where Pro2 is a single-processor node and Pro2 is a 2-processor node.

Table 4. 1 Comparison of the basic CFS and PS scheduler
(Pro2 is a single processor node)

		CFS		PS	
Test case	Task name	Utilization	Response time	Utilization	Response time
Case 1: m=1; n=1	TaskA	0.4921	2.0321	0.49536	2.0186
	TaskB	0.49613	2.0156	0.49695	2.0123
Case 2: m=10; n=3	TaskA	0.76792	13.02	0.76437	13.081
	TaskB	0.23109	12.978	0.23048	13.015
Case 3: m=20; n=5	TaskA	0.79989	24.999	0.80029	24.984
	TaskB	0.20196	24.753	0.19808	25.239

Conclusions:

The utilizations of the TaskA and TaskB depend on their multiplicities respectively. This

is because the basic CFS scheduler is fair to tasks. The basic CFS scheduler can get the same results as the PS scheduler, this shows that the basic CFS scheduler works correctly.

Table 4.2 Comparison of the basic CFS and PS scheduler
(Pro2 is a 2-processor node)

		CFS		PS	
Test case	Task name	Utilization	Response time	Utilization	Response time
Case 1: m=1; n=1	TaskA	0.89389	1.1187	0.89926	1.112
	TaskB	0.89901	1.1123	0.90705	1.1025
Case 2: m=10; n=3	TaskA	1.5438	6.477	1.549	6.4554
	TaskB	0.46037	6.5157	0.46166	6.4982
Case 3: m=20; n=5	TaskA	1.6033	12.473	1.5975	12.518
	TaskB	0.39852	12.545	0.40269	12.416

4.2 The group CFS scheduler

In this section, the group element is applied into the basic input model, and the group input model is generated. TaskA and TaskB are put into two separate groups, GroupA and GroupB. Figure 4.2 shows the group input model.

4.2.1 Comparing CFS with FCFS and PS

In this section, all the test cases are based on the same parameters described as follow.

1. The processor node, Pro2 is a single processor.
2. Shares of GroupA and GroupB are 0.75 and 0.25 respectively.
3. Multiplicity of UserA and UserB are 25.

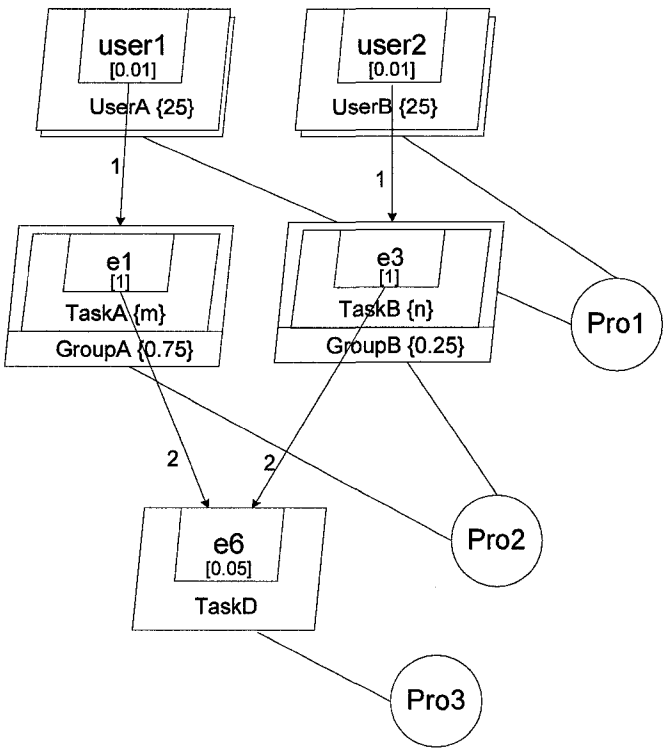


Figure 4.2 Group input model (for testing)

The following three test cases are performed using the FCFS, PS and group CFS schedulers. Tables 4.3 and 4.4 show the results.

Table 4.3 The comparison of group CFS, FCFS and PS (TaskA and TaskB)

		CFS		FCFS		PS	
Test case	Task name	Utilization	Response time	Utilization	Response time	Utilization	Response time
Case 1: m=1; n=1	TaskA	0.72588	1.3776	0.49586	2.0168	0.49564	2.0176
	TaskB	0.27063	3.6951	0.4951	2.0309	0.49791	2.0084
Case 2: m=10; n=3	TaskA	0.74984	13.334	0.76911	12.961	0.77175	12.956
	TaskB	0.24778	12.105	0.23089	13	0.23372	12.835
Case 3: m=20; n=5	TaskA	0.74723	26.757	0.7998	25.086	0.80235	24.922
	TaskB	0.24862	20.109	0.2002	25.039	0.20103	24.864

Table 4.4 The comparison of group CFS, FCFS and PS (UserA and UserB)

		CFS		FCFS		PS	
Test case	Task name	Through-put	Response time	Through-put	Response time	Through-put	Response time
Case 1: m=1; n=1	UserA	0.72647	34.395	0.50074	49.881	0.49639	50.32
	UserB	0.26784	93.176	0.48834	51.152	0.49363	50.608
Case 2: m=10; n=3	UserA	0.75203	33.226	0.77405	32.281	0.76631	32.613
	UserB	0.24853	100.46	0.23073	107.3	0.23401	106.64
Case 3: m=20; n=5	UserA	0.75134	33.254	0.78865	31.678	0.80971	30.859
	UserB	0.24235	103.04	0.19858	125.66	0.20402	122.38

Conclusions:

1. The effects of group shares act in the expected way. The utilizations of TaskA and TaskB are very close to their predefined group shares regardless the number of the copies of TaskA and TaskB.

2. Group shares have no effect on the FCFS and PS schedulers. These two schedulers are fair to tasks.

3. The utilizations of TaskA and TaskB are in proportion to the ratio of their multiplicity in the FCFS and the PR cases.

4. Using the group CFS scheduler, the response times of TaskA and TaskB are inversely related to their group shares. The ratio can be represented by

$$f_{share} = share_{GroupA} / share_{GroupB} .$$

Further, the response times of TaskA and TaskB are in proportion to the ratio of the number of copies as well. This ratio can be expressed by $f_{multiplicity} = m / n$.

5. Combining the two relationships, the ratio of the response time of TaskA and TaskB is equal to the result of $f_{multiplicity}$ / f_{share} . Table 4.5 shows the relationship.

Table 4.5 Relationship among response time, multiplicity and group share

	Case 1	Case 2	Case 3
the ratio of group share, f_{share}	3/1=3	3/1=3	3/1=3
the ratio of multiplicity, $f_{multiplicity}$	1/1=1	10/3=3.33	20/5=4
the result of $f_{multiplicity}$ / f_{share}	1/3=0.33	3.33/3=1.11	4/3=1.33
the ratio of response times, $f_{response\_time}$	1.3776/3.6951 =0.37	13.334/12.105 =1.10	26.757/20.109 =1.33

From the table, Case 2 and Case 3 of the CFS scheduling obey this relation. Case 1 does not, because the processor, Pro2, is not fully utilized.

6. Group shares only have partial effects in non-fully utilized processor.

4.2.2 Single-processor and multi-processor

In this part, the processing node, Pro2 can be either a single-processor node or a multi-processor node. The three test cases are similar to the previous section, but are run using the group CFS scheduler only. The results are shown in Tables 4.6 and 4.7.

Conclusions:

1. The results obey the same rules of Section 4.2.1.

2. The response times of TaskA and TaskB, are inversely related to the multiplicity of processor.

Table 4.6 The comparison of a single-processor and 2-processor (TaskA and TaskB)

		Pro2 is a single-processor node		Pro2 is a 2-processor node	
Test case	Task name	Utilization	Response time	Utilization	Response time
Case 1: m=1; n=1	TaskA	0.72588	1.3776	0.89884	1.1125
	TaskB	0.27063	3.6951	0.90047	1.1105
Case 2: m=10; n=3	TaskA	0.74984	13.334	1.5009	6.6622
	TaskB	0.24778	12.105	0.49637	6.0437
Case 3: m=20; n=5	TaskA	0.74723	26.757	1.4938	13.387
	TaskB	0.24862	20.109	0.49833	10.032

Table 4.7 The comparison of a single-processor and 2-processor (UserA and UserB)

		Pro2 is a single-processor node		Pro2 is a multi-processor(2) node	
Test case	Task name	Throughput	Response time	Throughput	Response time
Case 1: m=1; n=1	UserA	**0.72647**	**34.395**	**0.8888**	**28.118**
	UserB	**0.26784**	**93.176**	**0.89744**	**27.848**
Case 2: m=10; n=3	UserA	0.75203	33.226	1.4997	16.665
	UserB	0.24853	100.46	0.49774	50.191
Case 3: m=20; n=5	UserA	0.75134	33.254	1.4885	16.79
	UserB	0.24235	103.04	0.5025	49.714

3. If the processor is not fully utilized, the group shares only have a partial effect or no effect at all demonstrated by the case 1 of the two-processor node (in bold). Similarly the processor in Case 1 (single-processor) is not completely fully utilized (99%), so the results have a slight difference when compared with the others.

4.2.3 Comparison of different share combinations

All the test cases in this section are based on the same parameters described that follows.

1. The processor node, Pro2, is a single processor.

2. The multiplicity of UserA and UserB are 30.

3. The combinations of group shares are: 0.75 and 0.25; 0.4 and 0.6; and, 0.3 and 0.5.

In this part, five test cases are performed with the different group share combinations. The results are shown in Table 4.8.

Table 4.8 The comparison of different share combinations

		75-25 share combination		40-60 share combination		30-50 share combination	
Test case	Task name	Utiliza-tion	Response time	Utiliza-tion	Response time	Utiliza-tion	Response time
Case 1: m=1; n=1	TaskA	0.72238	1.3843	0.40377	2.4766	0.39998	2.5001
	TaskB	0.26819	3.7287	0.58746	1.7022	0.58643	1.7052
Case 2: m=10; n=3	TaskA	0.75036	13.326	0.39673	25.2	0.40129	24.482
	TaskB	0.25033	11.983	0.59853	5.012	0.59598	5.0335
Case 3: m=20; n=5	TaskA	0.75119	26.619	0.39995	49.971	0.39769	50.265
	TaskB	0.24909	20.071	0.60153	8.311	0.59875	8.3498
Case 4: m=10; n=10	TaskA	0.74763	13.379	0.40036	24.97	0.4007	24.954
	TaskB	0.24837	40.25	0.59412	16.83	0.60112	16.634
Case 5: m=10; n=20	TaskA	0.7549	13.244	0.4044	24.721	0.39733	25.161
	TaskB	0.25091	79.636	0.60364	33.115	0.59628	33.531

Conclusions:

1. In the 0.75-0.25 and 0.4-0.6 share combinations, group shares act in the normal way.

2. The most interesting case is the 0.3-0.5 share combination. Since the sum of group shares is less than 1, the groups can receive higher utilizations than their entitled shares. The results indicate they reach to the 0.4-0.6 share combination. That means that each group can get an identical portion of the surplus share regardless of how large or small its original share is. The extra amounts they get more are not in proportion to their shares; rather this case favors groups with small shares.

4.3 The mixed CFS scheduler

In this section, the definition of group shares is changed from the guarantee to the cap, and a comparison is performed between them. The test cases use the same model found in section 4.2.1. The shares of GroupA and GroupB are 0.5 and 0.3.

Tables 4.9 and 4.10 show the utilizations and response times of TaskA and TaskB respectively where Pro2 is both a single-processor and 2-processor node.

Conclusions:

1. Caps act in the expected way. They are able to control the utilizations of groups very well.

2. If the definitions of a cap and a guarantee exist in a same input LQN model, the spare share is allocated to the groups with guarantee shares.

3. If all group shares are defined by caps, the surplus share is wasted, no group can get it.

Table 4.9 The mixed CFS scheduler (Pro2 is a single processor node)

		The shares of GroupA and GroupB are guarantees		The share of GroupA is a guarantee, and the one of GroupB is a cap		The shares of GroupA and GroupB are caps	
Test case	Task name	Utilization	Response time	Utilization	Response time	Utilization	Response time
Case 1: m=1; n=1	TaskA	0.58745	1.7023	**0.66337**	1.5074	**0.47513**	2.1046
	TaskB	0.40667	2.4589	**0.28956**	3.4534	**0.29295**	3.4135
Case 2: m=10; n=3	TaskA	0.59957	16.676	**0.69924**	14.3	**0.49948**	18.94
	TaskB	0.40269	7.4492	**0.3**	9.5343	**0.29768**	9.4643
Case 3: m=20; n=5	TaskA	0.59973	33.34	**0.70367**	28.415	**0.50301**	37.541
	TaskB	0.39899	12.531	**0.29786**	16.082	**0.30047**	15.717

Table 4.10 The mixed CFS scheduler (Pro2 is a 2- processor node)

		The shares of GroupA and GroupB are guarantees		The share of GroupA is a guarantee, and the one of GroupB is a cap		The shares of GroupA and GroupB are caps	
Test case	Task name	Utilization	Response time	Utilization	Response time	Utilization	Response time
Case 1: m=1; n=1	TaskA	0.90169	1.109	0.78178	1.2791	0.47514	2.1046
	TaskB	0.9008	1.1101	0.28901	3.4601	0.29013	3.4466
Case 2: m=10; n=3	TaskA	1.1997	8.3348	1.447	6.9105	0.97737	9.8534
	TaskB	0.79698	3.764	0.55404	5.3598	0.51228	5.8324
Case 3: m=20; n=5	TaskA	1.1978	16.696	1.4266	14.017	0.98503	19.371
	TaskB	0.79809	6.2645	0.57477	8.5445	0.54972	9.0157

4.4 Computation time

The model in Section 4.2.1 (case 3) is used, whose parameters are:

1. The processor node, Pro2, is a single processor.

2. Shares of GroupA and GroupB are guarantees and are 0.75 and 0.25 respectively.

3. Multiplicity of UserA and UserB are 25.

4. Multiplicity of TaskA and TaskB are 20 and 5 respectively.

5. Quantum is 0.1 ms for the PS and CFS scheduler.

The model is run twenty times continuously for each scheduler, the total computation times of the CFS, PS and FCFS schedulers are 40, 22 and 18 seconds respectively. The definitions of groups and shares do not affect the PS and FCFS schedulers while they do affect the CFS scheduler. The mixed CFS scheduler is the slowest, the group CFS scheduler is the next, and the basic CFS scheduler is the fastest. The value of quantum also affects the computation times of the PS and CFS schedulers.

4.5 Conclusion

The results of the testing show that the PARASOL CFS scheduler integrates the basic CFS scheduler, group CFS scheduler and Mixed CFS scheduler function correctly.

Chapter 5: Case Study

The aim of this case study is to explore how the CFS scheduler impacts the system behavior and the performance of an application system. The application system of the case study is a Building Security System (BSS) described by Jing Xu in [37]. In [37], this system was studied by using a LQN performance model to analyze and improve its performance and satisfy the non-functional requirements defined by the UML SPT profile.

The purpose of the case study here is to use the group CFS and mixed CFS schedulers to control the utilization of the tasks in the system and improve the performance of the entire system. The case study will show the effect of the CFS scheduler through studying the interrelationship of the two sub-systems found in the BSS model.

This chapter consists of three parts. In the first part, the BSS system is described and some pre-experiments are performed to establish base-line performance results. The second part introduces the performance problem first, then applies the CFS scheduler to one of the processor nodes, ApplicCPU, to address the problem. In the last part, another set of experiments are applied to the DB_CPU.

5.1 The LQN model of the BSS

The BSS system is a model of an access control and security system for a commercial building. It consists of two major components: access control and video surveillance. Access control is in charge of reading a user's card, comparing the information with a database, and opening the door or ringing an alarm. The functionality of the video surveillance component consists of getting images from a set of cameras, then buffering and storing the images in a database. The two functions are referred to user part and video part for simplifying the discussion that follows. The deployment of the system is shown in Figure 5.1.

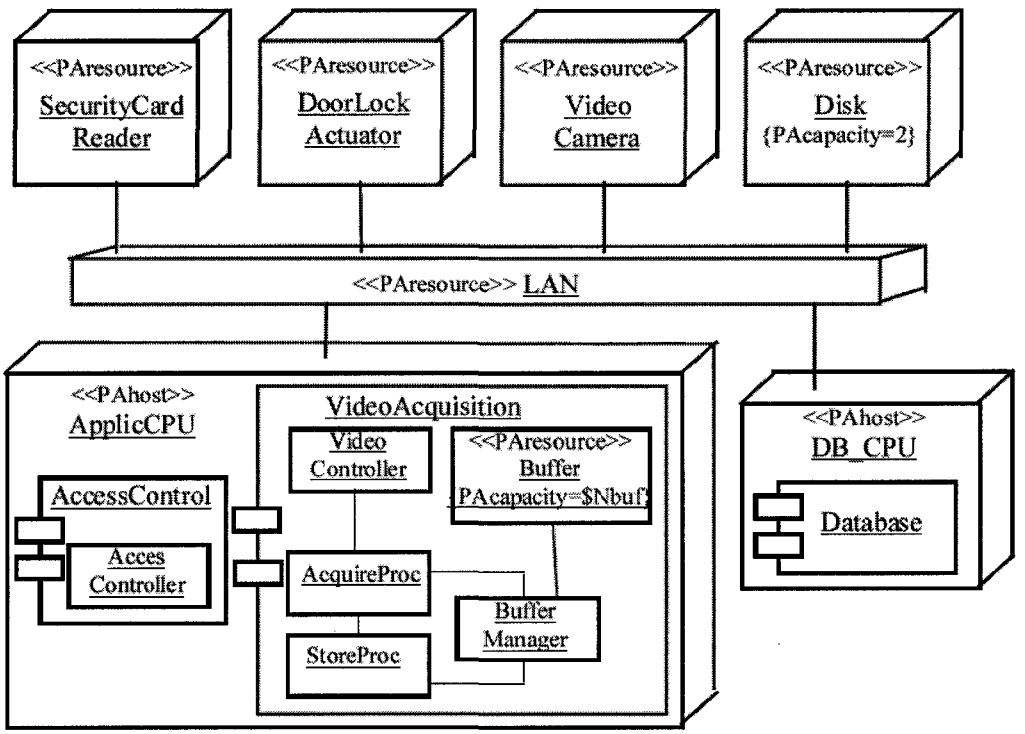


Figure 5.1 Deployment of the Building Security System [37]

All the peripheral devices can be reached by a LAN. According to the design of this system, there is an individual processor for database operations which is separated from an application processor which controls and runs the entire BBS system, and there are two disks in the BSS.

The model used for this case study is slightly different from the model described in [37]. The starting point is called the basic LQN model, and is shown in Figure 5.2. This is a closed LQN model with two classes, user and video. User_T and VidCtrl_T are the two reference tasks driving the two classes, with 50 ms and 0 ms think of time respectively. In the figure, $N stands for the number of cameras, and also represents the capacity of the video component of the BSS. VidCtrl_T task has no thinking time which means that it will deal with images from each camera continuously. The difference from the model in [37] is that the user class was an open class. Using a closed model for this class simplifies the analysis because the system will always be driven to its maximum capacity.

These two classes interact with each other through competition for the processor, ApplicCPU. The main targets of the study are the processor ApplicCPU, and the tasks competing for this processor. These tasks are AccCtrl_T, VidCtrl_T, StoreProc_T, PassImg_T, GetImg_T and AcqProc_T.

5.2 Pre-experiment

The main purpose of the pre-experiment is the determination of appropriate parameters that are suitable for the experiments which follow.

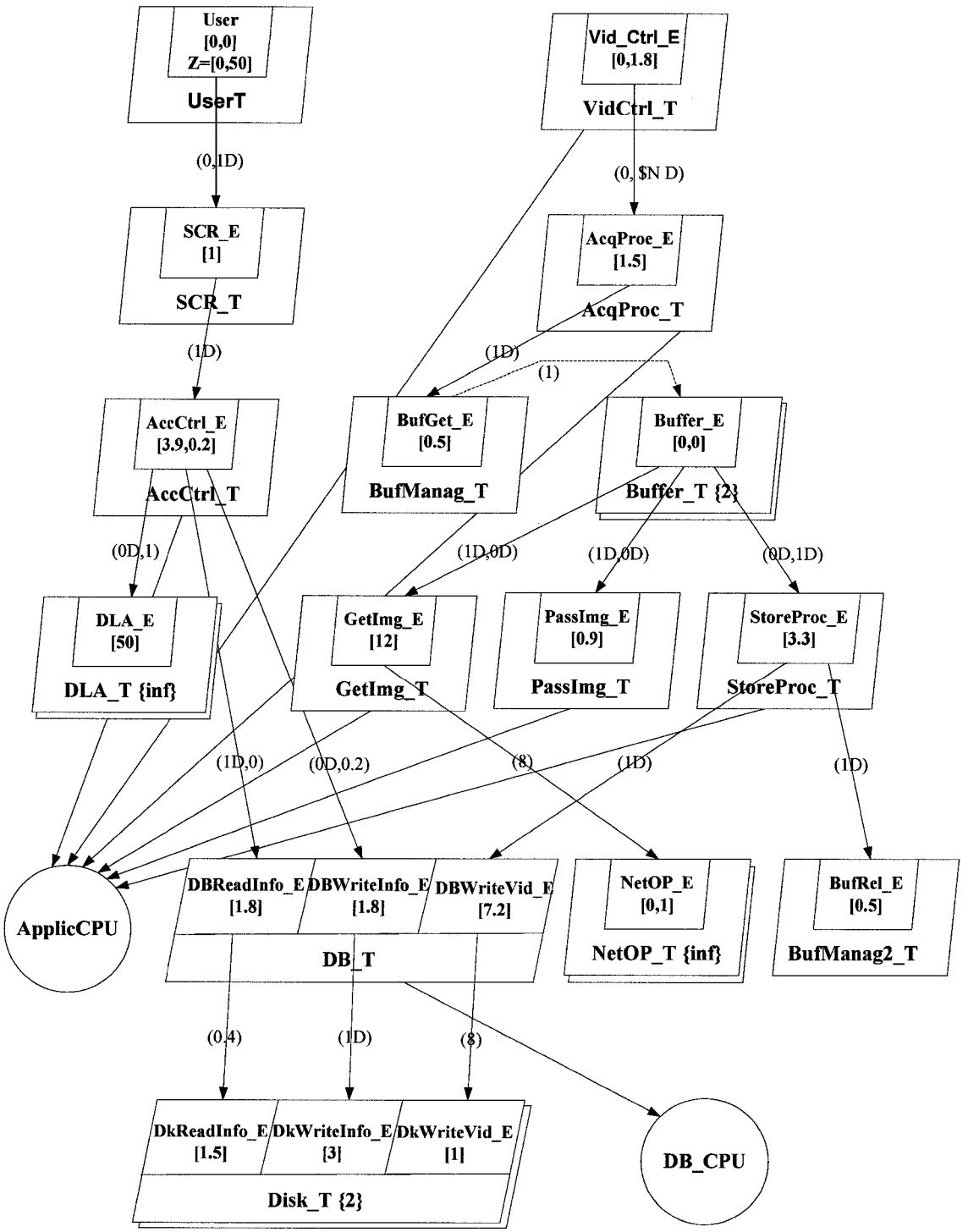


Figure 5.2 The Basic LQN model (Unit : ms)

5.2.1 Part 1: Video component capacity

The first part of the pre-experiment is adjusting the multiplicity parameters of all the tasks to get the proper capacity ($N) of the video part. These experiments were performed by increasing the number of cameras from 1 to 90, while the number of users is fixed at 1. The processor ApplicCPU is a single processor node.

In the experiments, as the value of $N increases, several tasks, such as Buffer_T, GetImg_E, StoreProc_E, and AcqProc_T, are saturated one or more times. These bottlenecks are software bottlenecks [11], so the multi-threading technique is applied to these saturated tasks to remove these software bottlenecks.

From the experiments, the utilizations of the six tasks on the processor ApplicCPU are collected, and shown in Table 5.1. The processor ApplicCPU is very close to being saturated when the number of cameras is over 70. For the rest of this study, the capacity ($N) was fixed at 70. At this level, the multiplicity parameters of the tasks, Buffer_T, StoreProc_T, GetImg_T and AcqProc_T were found to be satisfactory at 10, 6, 6, and 6 repectively.

Table 5.1 The Utilizations of the tasks on Processor ApplicCPU
(The number of users is 1)

$N / Task Name	Utilizations				
	1	10	20	30	40
Processor ApplicCPU	0.75775	0.74630	0.75686	0.82437	0.84104
Video Part	0.71851	0.70964	0.71828	0.78357	0.80082
User Part	0.03924	0.03665	0.03858	0.04080	0.04022
$N / Task Name	Utilizations				
	50	60	**70**	80	90
Processor ApplicCPU	0.95769	0.971439	**0.985102**	0.980583	0.985558
Video Part	0.91730	0.929578	**0.942298**	0.940526	0.943164
User Part	0.04039	0.041861	**0.042804**	0.040057	0.042394

The response time and throughput of User_T and VidCtrl_T are presented in the following Figures 5.3 and 5.4.

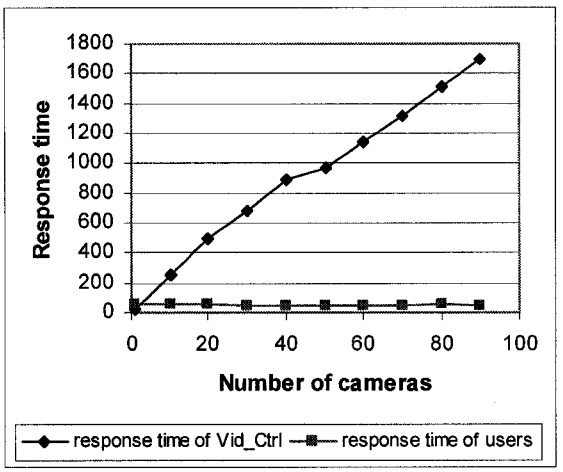


Figure 5.3 The response time of the video part and user part in Pre-experiment1

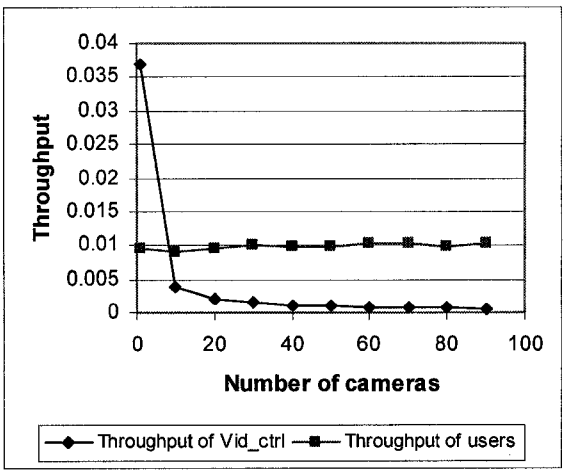


Figure 5.4 The throughput of the video part and user part in Pre-experiment1

5.2.2 Part 2: User component capacity

The second part of the pre-experiment is used to find enough multiplicity of the tasks in the user component. This will ensure that the tasks in the user part will not be software bottlenecks in the experiments that follow. The saturated device will be the processor ApplicCPU only, and it will become fully utilized.

The value of $N is fixed at 1 for this part. The process and method are similar to the first part of the pre-experiment. The result is that the processor ApplicCPU is saturated when the number of users approaches 30. The multiplicity parameters of AccCtrl_T and SCR_T are found to be 60 and 50 respectively. In addition, when combining these parameters with the first part, the multiplicity of DB_T is 10. These parameters will

ensure that there is no software bottleneck in the user part if the number of users does not exceed 50. The multiplicity parameters of these tasks are shown in Table 5.2.

Table 5.2 The multiplicity parameters of tasks

Task name	multiplicity
VidCtrl_T	1
StoreProc_T	6
AccCtrl_T	60
PassImg_T	1
Buffer_T	10
GetImg_T	6
SCR_T	50
DB_T	10
AcqProc_T	6

5.3 Experiment 1: Using the CFS group scheduler in ApplicCPU

This set of experiments includes the following parts:

1. Experiment 1-1: use PS scheduling at the processor ApplicCPU to illustrate what will happen to the tasks in the video part as the number of users increases. Will the performance of the video component be adversely affected?

2. Experiment 1-2: apply the CFS scheduler to the processor ApplicCPU to improve the performance of the video component.

3. Experiments 1-3 and 1-4: try some different group strategies to refine the results.

5.3.1 Experiment 1-1: Using PS scheduling

Based on the results of the pre-experiments, PS scheduling is used to find the interrelationship of the user part and the video part. The capacity of the video part is

fixed at 70 and the number of users is increased from 1 to 50. The results of the experiments can be illustrated through the changes of the response times, throughputs and utilizations of User_T and VidCtrl_T. The processor ApplicCPU is saturated when the number of users reaches 3.

For the user part, as the number of users increases, the utilization of the task AccCtrl_T increases quickly although the processor is saturated already. AccCtrl_T gets more CPU resources by gradually driving the video part away. When the number of users is 20, the user part is saturated. The response time, throughput and utilization of users are shown in Figures 5.5, 5.6 and 5.9 respectively.

For the video part, as the number of users increases, its utilization decreases quickly. When the number of users reaches 20, the share of the CPU for the video part is nearly driven away. After 20 users, the response time of VidCtrl_T increase significantly and is hundreds of times more than one of users, while both the throughput and utilization of VidCtrl_T approach zero. Figures 5.7, 5.8 and 5.9 show the results.

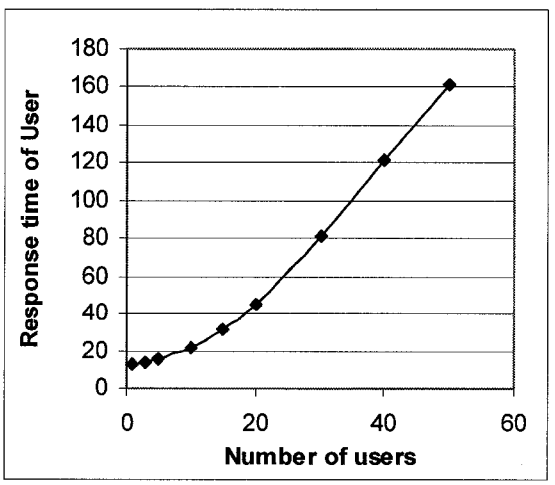


Figure 5.5 The response time of the user part in Experiment 1-1

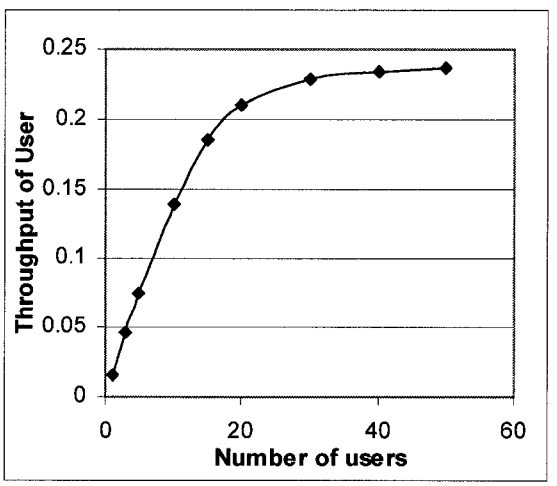


Figure 5.6 The throughput of the user part of in Experiment 1-1

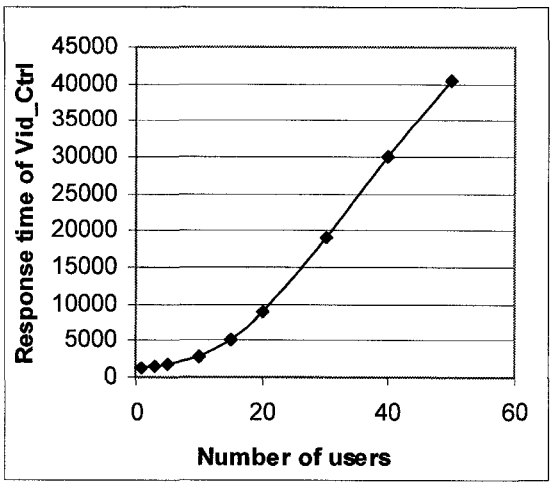


Figure 5.7 The response time of the video part in Experiment1-1

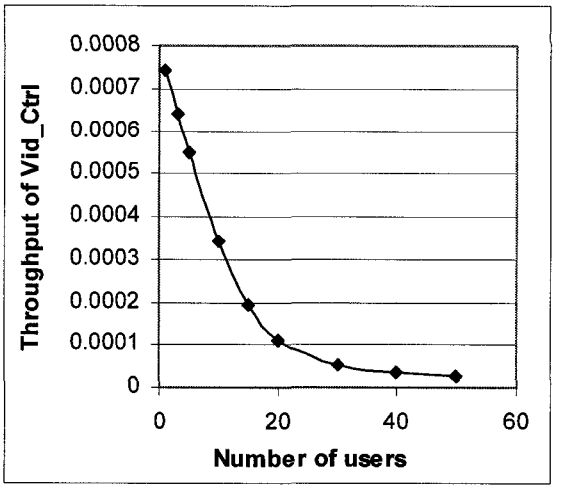


Figure 5.8 The throughput of the video part in Experiment1-1

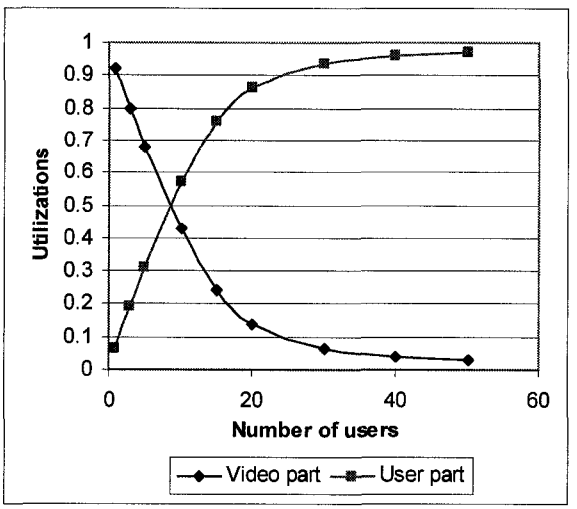


Figure 5.9 The utilizations of the user part and video part in Experiment1-1

The results are not surprising, but are unacceptable for the BSS or any other similar applications. According to the requirements and functionality of the BSS, the video part should function normally regardless of the number of users. One solution that can be used is to use a multi-processor for ApplicCPU to improve the situation temporarily. But the problem still exists as the number of users increases further. The CFS scheduler will be used at the ApplicCPU to solve the problem.

5.3.2 Experiment 1-2: Applying the CFS group scheduler

In this stage of the experiments, the input model has the same parameters as Experiment 1-1, except that the group definition and the CFS group scheduler are applied. This new LQN model is called the basic group model, to differentiate it with the group models in the later experiments. The groups in the basic group model are represented by the new group notations, and are illustrated in Figure 5.10. The goal of the experiment is to maintain 80% utilization of the ApplicCPU for the video part regardless of the number of users. The key questions are: how to group the tasks and how much should their shares be.

In this experiment, the six tasks are grouped into six groups. The information from the data in Section 5.3.1 is used for choosing the values of the shares, shown in Table 5.3.

Table 5.3 Utilizations of tasks on the processor ApplicCPU

Number of Users	1	3	5
Processor	Utilization		
VidCtrl_T	0.001314	0.001134	0.001008
StoreProc_T	0.17048	0.14705	0.13042
PassImg_T	0.046494	0.040122	0.035568
GetImg_T	0.61992	0.53496	0.47424
AcqProc_T	0.07749	0.06687	0.05928
Sum of video part	0.915698	0.790136	0.700516
AccCtrl_T	0.065354	0.19446	0.30984
Total of ApplicCPU	0.981052	0.984596	1.010356

The utilizations for the case for three users are chosen as the shares of the six groups. Table 5.4 shows the group information. The total share of the groups in the video part is 0.8.

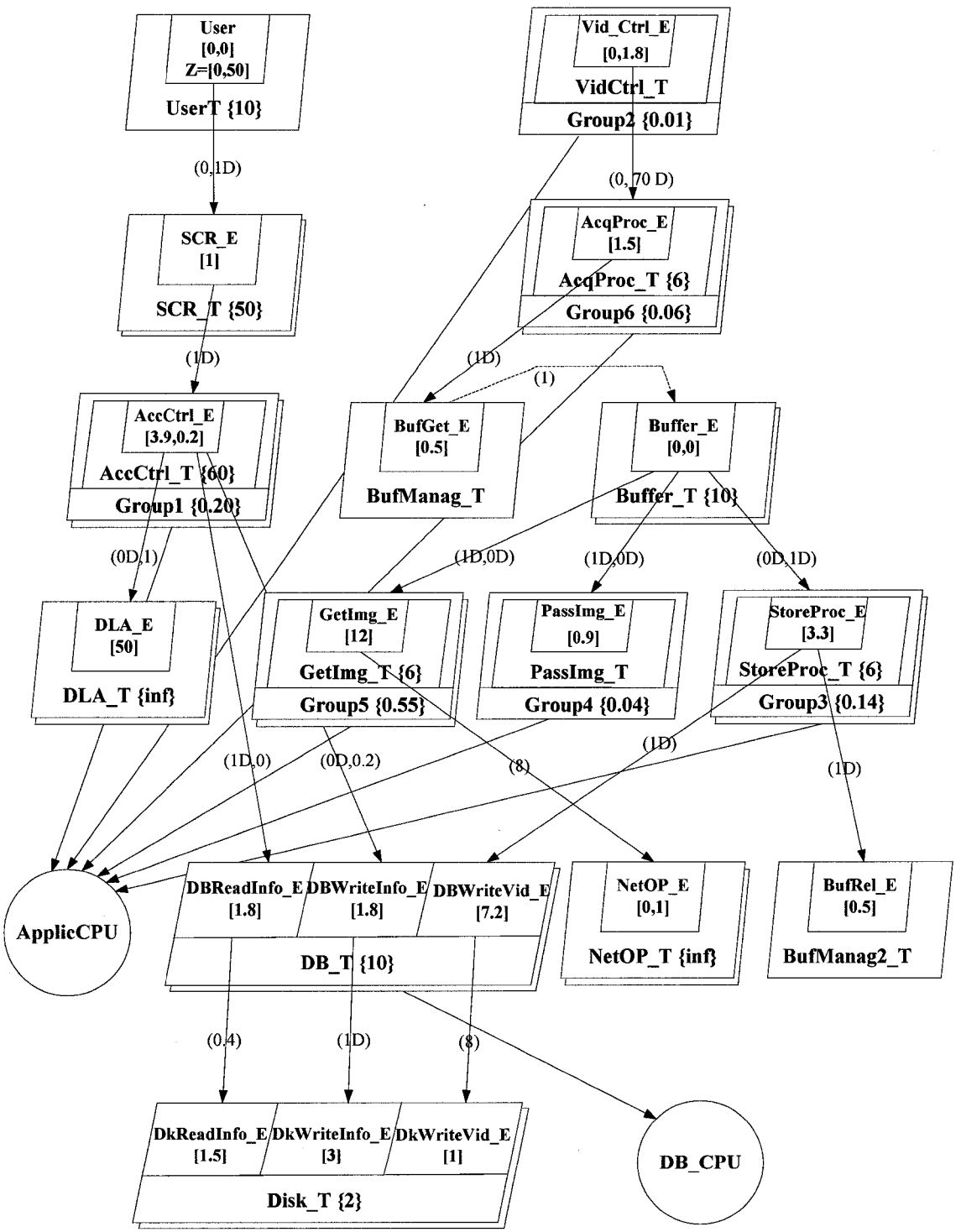


Figure 5.10 The basic group model

Table 5.4 Group information

Task Name	Group name	Group share
AccCtrl_T	Group 1	0.20
VidCtrl_T	Group 2	0.01
StoreProc_T	Group 3	0.14
PassImg_T	Group 4	0.04
GetImg_T	Group 5	0.55
AcqProc_T	Group 6	0.06

The results of the experiments are shown in the following five figures, which compare the CFS group scheduler to the PS scheduler. For the user part, Figure5.11 shows that the response time of users will increase rapidly after the ApplicCPU is saturated because the group AccCtrl_T of the user part does not have an enough share to satisfy the growing number of users. Therefore, after group AccCtr_T runs out its share, the user part becomes saturated. Figure 5.12 shows the same result, the throughput of users can not go up after three users.

For the video part, Figure 5.13 shows that the response time of VidCtrl_T keeps a fixed value after the ApplicCPU is saturated. The shares of groups in the video part, make the video part perform its normal functionality at a certain level rather than be driven away as the number of users grows as was shown earlier. Similarly, Figure 5.14 shows that the throughput of the task VidCtrl_T plateaus with CFS scheduling while it goes down to zero in the PS scheduling case. Figure 5.15 shows the changes of the utilizations of the user part and the video part on the processor ApplicCPU.

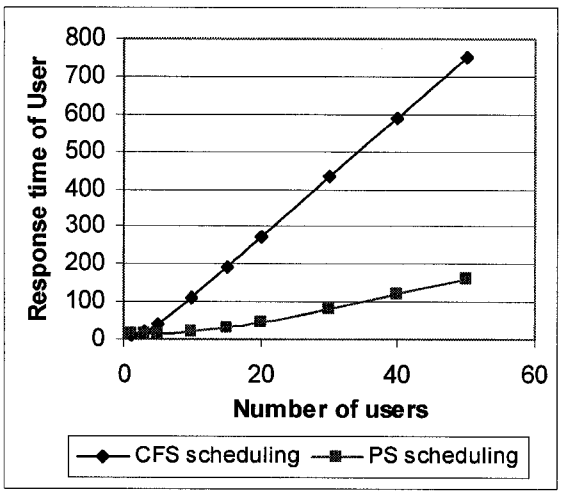


Figure 5.11 The response time of the user part comparing the CFS and PS scheduling

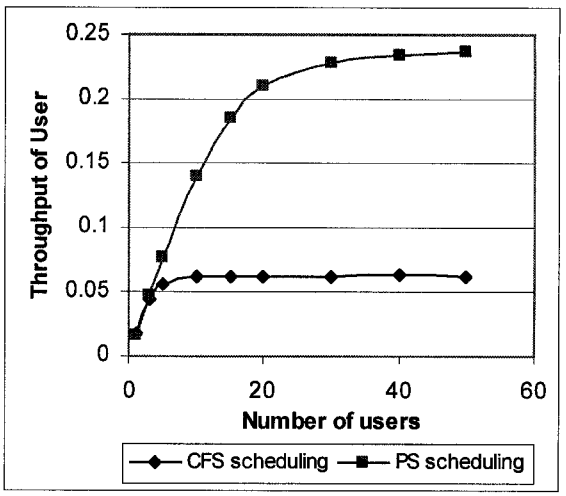


Figure 5.12 Throughput of the user part comparing the CFS and PS scheduling

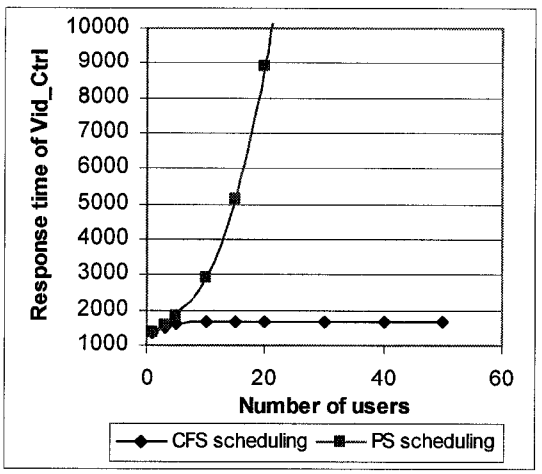


Figure 5.13 The response time of the video part comparing the CFS and PS scheduling

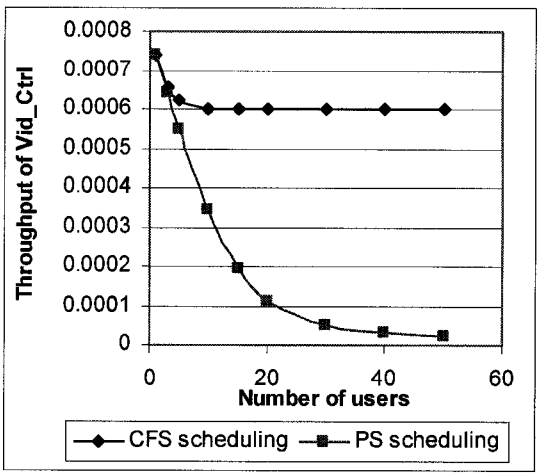


Figure 5.14 Throughput of the video part comparing the CFS and PS scheduling

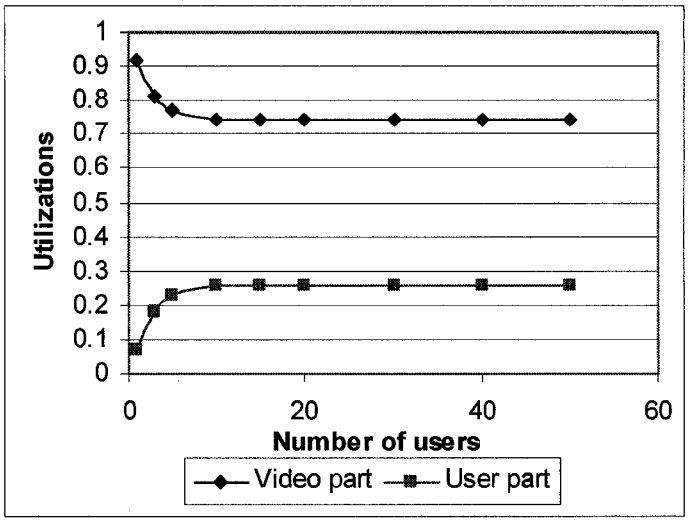


Figure 5.15 The utilizations of user part and video part in Experiment 1-2.

This experiment demonstrates the CFS group scheduler has the desired effect. The group share of the user part limits the utilization of the tasks in the user part. This can protect the video part from being starved of CPU resources.

Although this experiment works, the result is not very satisfactory. The total share of the video part is 80%, but the video part only got a 74% of the utilization. Therefore, the group of AccCtrl_T obtained 6% more than its entitled share. There are two reasons for this result. The first one is that the share combination is not proper. The second reason has to do layered interactions found in the BBS model. The database task DB_T is a shared resource to both the video and user components of the system. Therefore, when the video components are blocked on the database, the user components can run. When using guarantee shares, this effect was demonstrated because the user component got more CPU than its shares. Experiment 3 in Section 5.5 will attempt to rectify the issue by applying CFS scheduling at the database processor.

5.3.3 Experiment 1-3: Different group share combinations

In this section, the group share of the user part is fixed at 0.2, and the number of users is fixed at 30. The group share combinations are shown in the first half of Table 5.5. The task GetImg_T is a task with the largest host demand, and it is in the lowest layer. So the group of GetImg_T is treated as a variable, and its share is increased from 0.16 to 0.7. The shares of the other groups in the video part are the same and decreased from 0.16 to 0.025. The utilizations of the video part and the user part are shown in the second part of Table 5.5.

Table 5.5 Different group share combinations (The number of users is 30)

Group Name	Group share Combinations				
	Case 1	Case 2	Case 3	Case 4	Case 5
AccCtrl_T	0.20	0.20	0.20	0.20	0.2
VidCtrl_T	0.16	0.125	0.1	0.05	0.025
StoreProc_T	0.16	0.125	0.1	0.05	0.025
PassImg_T	0.16	0.125	0.1	0.05	0.025
GetImg_T	**0.16**	**0.3**	**0.4**	**0.6**	**0.7**
AcqProc_T	0.16	0.125	0.1	0.05	0.025
Utilization of the user part	0.44251	0.37043	0.31562	0.2845	0.31082
Utilization of the video part	0.56319	0.62111	0.68343	0.70835	0.69808

The utilization of the video part increases as the group share of the group GetImg_T increases from 0.14 to 0.6. The results of the experiment indicate that a group with a task with a large service demand deserves a large share, especially if this task is in a lower layer. The reason is that the large share ensures that the task finishes quickly, and it reduces the blocking time to tasks in upper layers. As a result, the performance of the video part is improved. In general, if a group is assigned a small share, the tasks in this

group are slowed down. Figure 5.16 shows that the response time of the group GetImg_T decreases as its share increases.

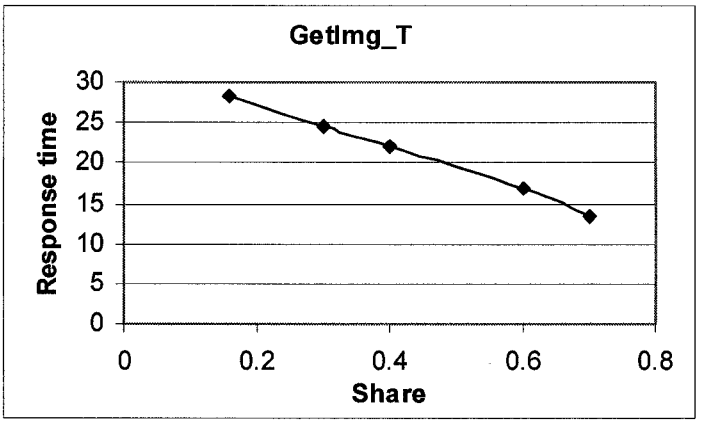


Figure 5.16 The response time of the group GetImg_T in Experiment 1-3.

In a multi-layered distributed system there are interactions between tasks, and the response times of the tasks in upper layers include the blocking time of the tasks in lower layers. This factor may result in the response time of upper layer tasks not being controlled by their group shares. Figures 5.17 and 5.18 show the response times of tasks VidCtrl_T and AcqProc_T. Although the group shares of these two groups are increased, the response time of these two groups are increased rather than decreased. The reason is that the group share of GetImg_T is decreased and the blocking time is increased as a result.

In Figures 5.17 and 5.18, the response time curves first decrease slightly then increase as expected. In this case (Case 5), the group share of GetImg_T is 0.7, and its response time reduced. However, the group shares of other groups in the video part are 0.025, and the response times of StoreProc_T and PassImg_T are increased. The increase of blocking

time of these two tasks is responsible for the increase of the response times of VidCtrl_T and AcqProc_T.

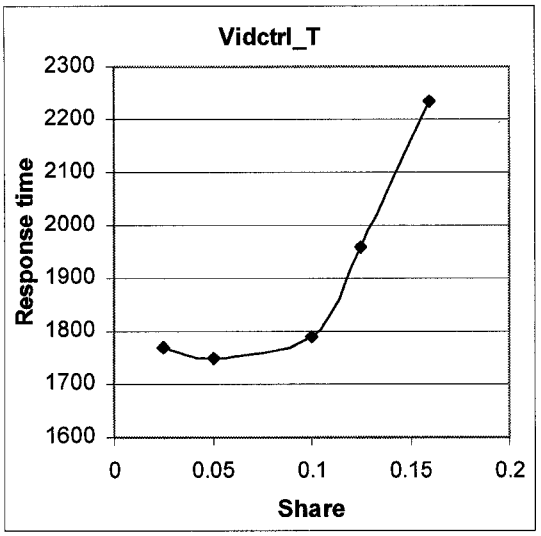


Figure 5.17 The response time of the group VidCtrl_T in the Experiment 1-3.

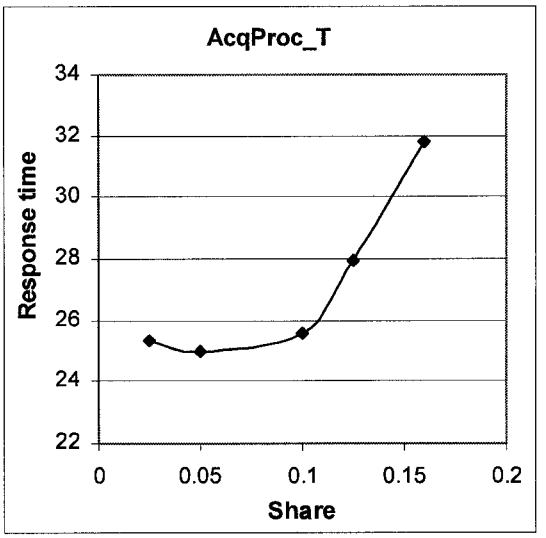


Figure 5.18 The response time of the group AcqProc_T in the Experiment 1-3.

But the result does not reach the goal, which is the utilizations of the video part and the user part of 0.8 and 0.2 respectively.

5.3.4 Experiment 1-4: Different group combinations

In this experiment, the six tasks are grouped into two or three groups only, to illustrate that the grouping is very important to get the expected results.

Case 1: two groups. AccCtrl_T is in group1 with a share of 20 %, while the remaining five tasks are in group2 with a total share of 80%. The result is shown in Figure 5.19. The utilization of the user part is 0.44, and for the video part is 0.56.

Case 2: three groups. AccCtrl_T is in the group1 with a share of 20 %, GetImg_T is in group2 with a share of 60%, and the remaining four tasks are in group3 with a total share

of 20%. The results are shown in Figure 5.20. The utilization of user part and video part are 0.287 and 0.712 respectively.

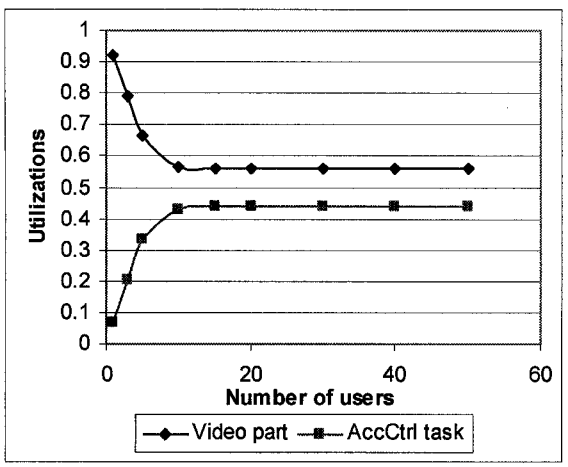


Figure 5.19 The utilizations of Case 1 in Experiment 1-4.

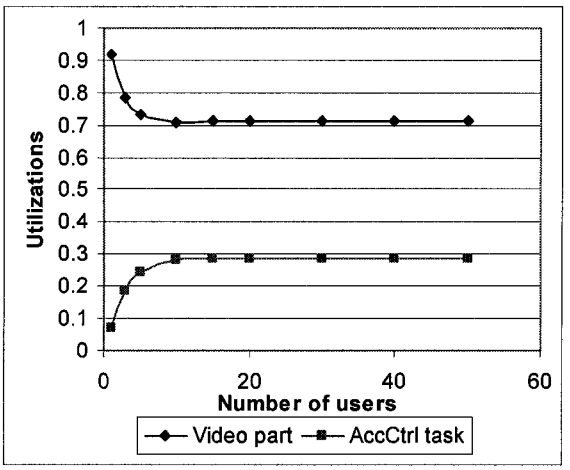


Figure 5.20 The utilizations of Case 2 in Experiment 1-4.

From the results of the two experiments, Case 2 is better than Case 1, but they do not satisfy the requirements. All the experiments above indicate the group shares need to be specified exactly. The question is whether the two parts can be grouped into two groups only, and get the expected result? The answer is positive. The mixed CFS scheduler is applied to the ApplicCPU processor.

5.4 Experiment 2: Apply the mixed CFS scheduler

The conditions of the experiment are the same as the Case 1 in Experiment 1-4, except that the share of the group1 is defined as a cap. So the group of AccCtrl_T has a cap share of 20%, and the group of the video part has a guarantee share of 80%. Figures 5.21 and 5.22 show the results.

The cap share acts in the expected way, which limits the utilization of the group AccCtrl_T to 20% strictly. The response time of users increases faster than the previous case. For this experiment, multiple runs were conducted to find confidence intervals for the result. The 95% confidence interval for the response times were no worse than ±1.8, and do not overlap.

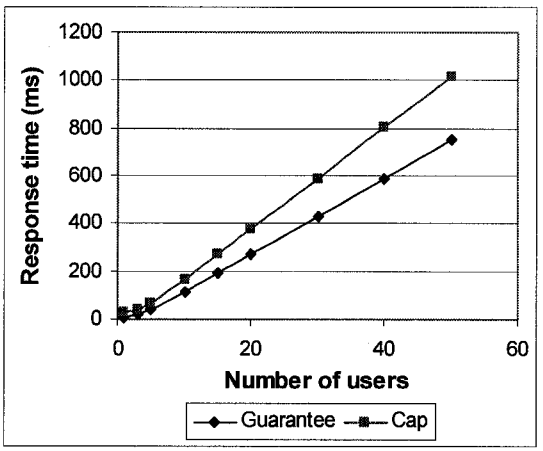


Figure 5.21 The response time of the user part comparing guarantee and cap shares

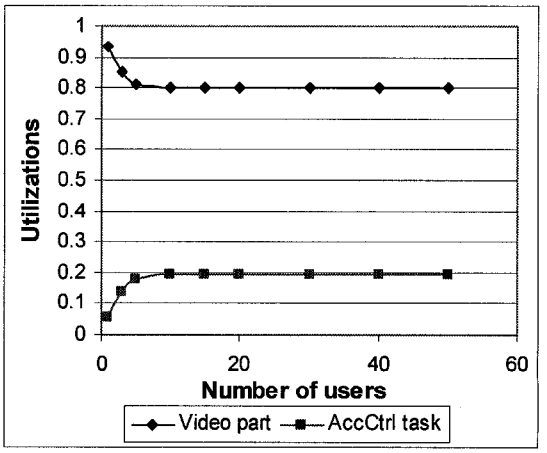


Figure 5.22 Utilizations of the user part and the video part in Experiment 2

5.5 Experiment 3: Applying the CFS scheduler to DB_CPU

In the BBS system, both user and video component block on the shared DB resource. In order to improve the performance of the system further, CFS scheduling will be performed at DB_CPU rather than at the ApplicCPU.

In order to perform the experiment, some modifications are made to the model first. There is only one task which is in charge of the database operations in the previous models. Therefore, the original DB_T task is separated into two tasks, DB_Info_T and DB_Vid_T, which are responsible for the database operations of the user part and the

video part respectively. Two models, the basic DB model and DB group model are generated, and they are used in Experiment 3. The basic DB model is presented in Figure 5.23. The method and the process of the Experiment 3 are similar to the Experiment 1.

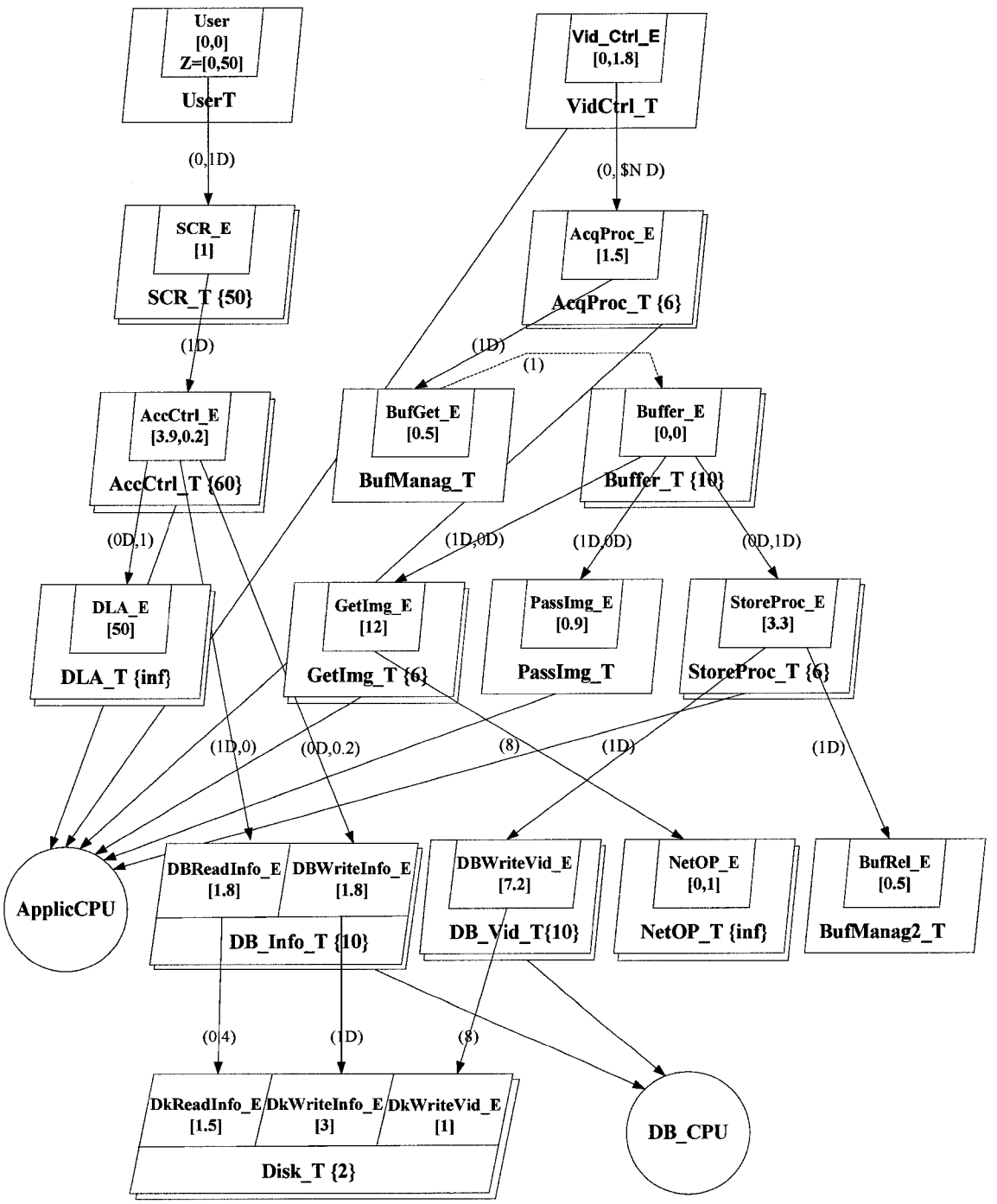


Figure 5.23 Basic DB-model

5.5.1 Experiment 3-1: Using the PS scheduler to get the utilizations of two components

In order to saturate the processor DB_CPU, the processor ApplicCPU is changed to a multi-processor node and its multiplicity is set to 3. Further, applying the second phase to the task AcqProc_T, and by increasing the copies of several tasks, this remove the other bottlenecks and saturate the processor DB_CPU. Figures 5.24 and 5.25 show the utilizations of the video part and the user part, and the task DB_Vid and the task DB_Info.

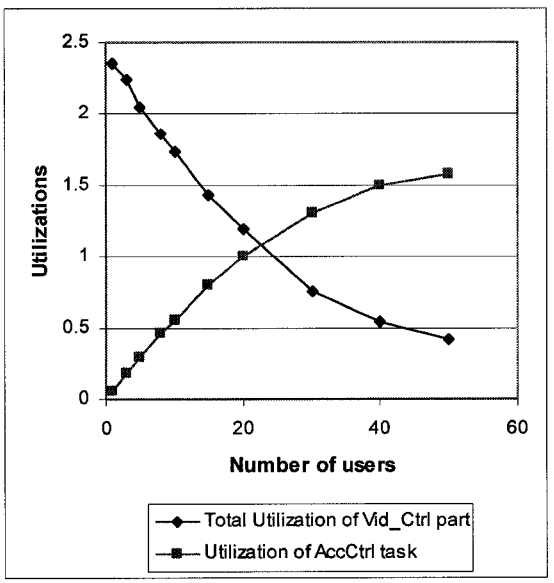


Figure 5.24 The Utilizations of the user part and video part in Experiment 3-1.

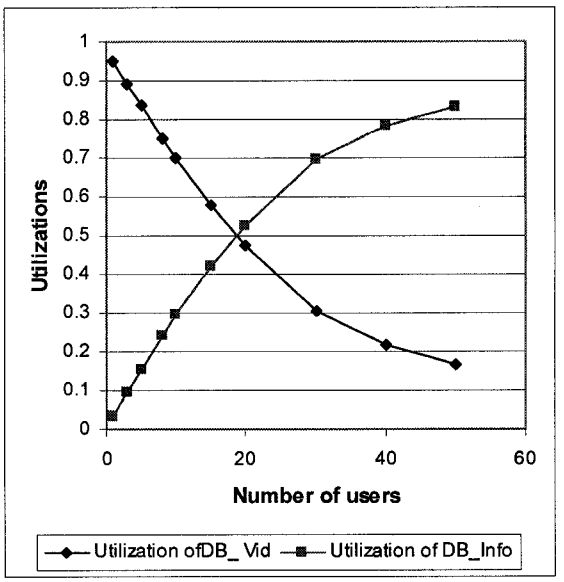


Figure 5.25 Utilizations of DB tasks of the user part and video part in Experiment 3-1.

As the number of users increases, the utilization of the user part increases, as does the utilization of the task DB_Info. Conversely, the utilization of the video part and the task DB_Vid decrease.

5.5.2 Experiment 3-2: Applying the CFS group scheduler

The goal of the experiment is keeping the utilization of the video part at 80%. The task DB_Info_T and DB_Vid_T are grouped into two groups. Their group shares can be obtained from Table 5.6. The DB group model is shown in Figure 5.26.

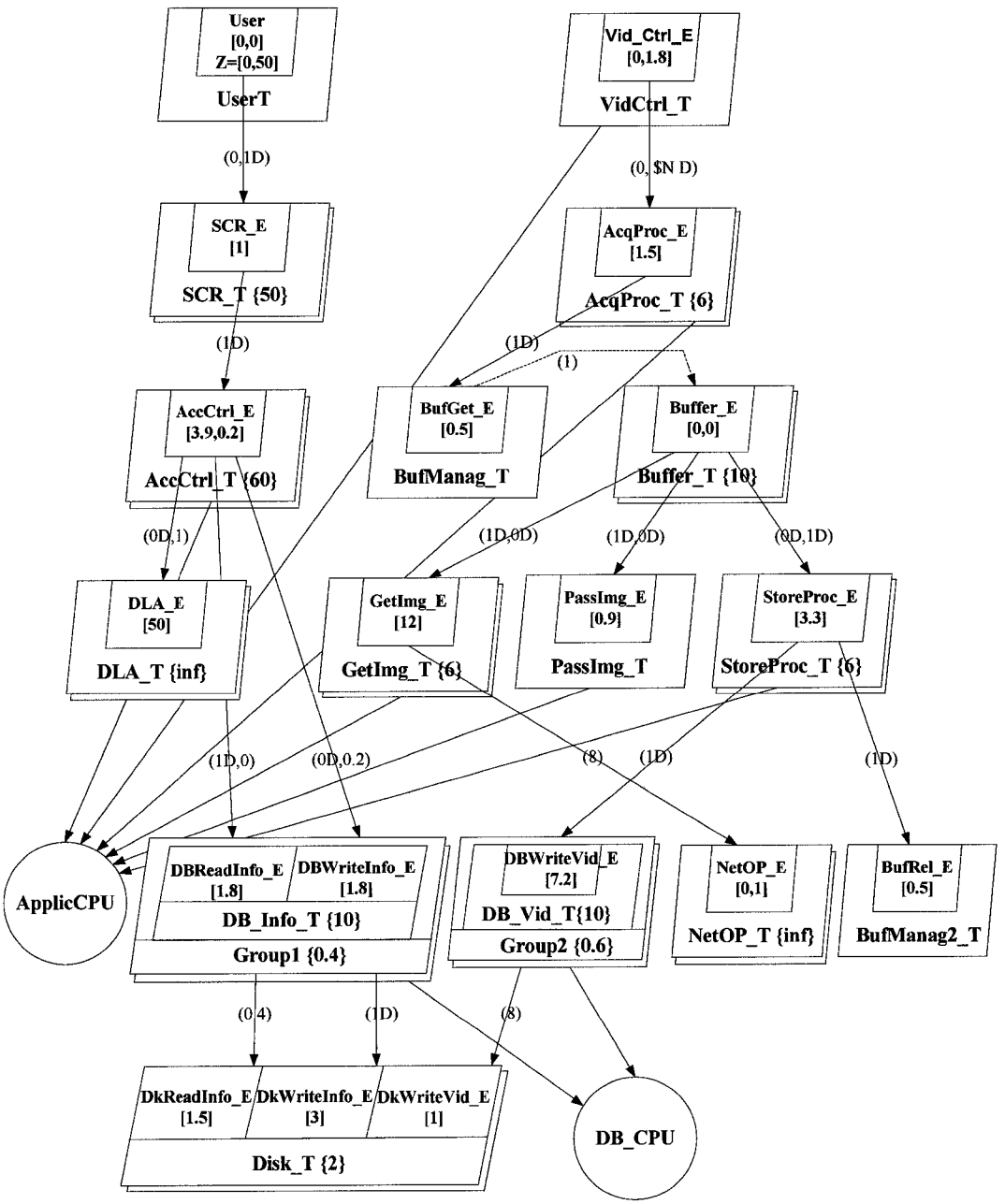


Figure 5.26 The DB group model

Table 5.6 The utilization of the user part and video part according to the DB tasks

Number of users	5	8	10	15
AccCtrl_T	0.29192	0.45928	0.55596	0.79565
Total Utilization of Video part	2.05494	1.853821	1.732049	1.42819
Total Utilization of ApplicCPU	2.34686	2.313101	2.288009	2.22384
Ratio of user part and video part	12.5/87.5	20/80	24/76	36/64
Utilization of DB_Info	0.15529	0.241993	0.294181	0.419898
Utilization of DB_Vid	0.83817	0.75168	0.70167	0.57907
Total Utilization of DB_CPU	0.99346	0.993673	0.995851	0.998968

When the number of users is 8, the utilizations of the user part and the video part are 20% and 80% on the processor AppliCPU. Therefore, the group shares of DB_Info and DB_Vid are 0.24 and 0.76 respectively. Figures 5.27 and 5.28 show the results.

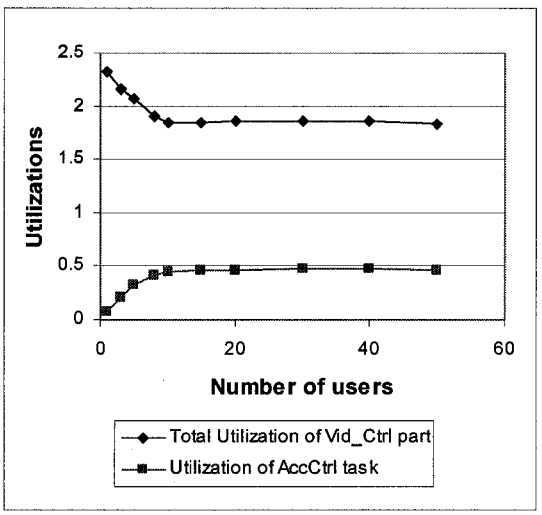


Figure 5.27 Utilizations of the user part and video part in Experiment 3-2

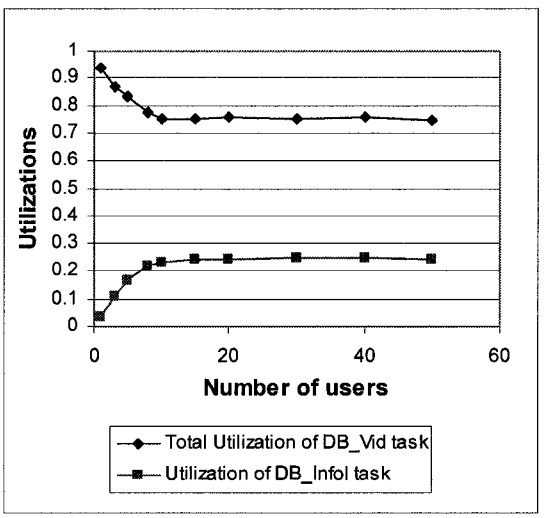


Figure 5.28 Utilizations of DB_Vid and DB_Info in the Experiment 3-2

In the experiment, the shares of groups limit the utilization of the group DB_Info and DB_Vid, as a consequence, the utilizations of the user part and the video part reach the ratio of 20/80. The result reaches the goal of the requirements.

The last experiment of this section, Experiment 3-3 applies the mixed CFS scheduler to DB_CPU by defining the share of DB_Info as a cap. The results are shown in Figure 5.29 and Figure 5.30. The result of the experiment is same as the result of the Experiment 3-2. The reason that caused the result to be the same is that the tasks DB_Info and DB_Vid are in the same layer. Therefore, if the tasks in groups are in the same layer, the guarantee shares have the same effect with the cap shares. Otherwise, guarantee shares may not act as well as expected.

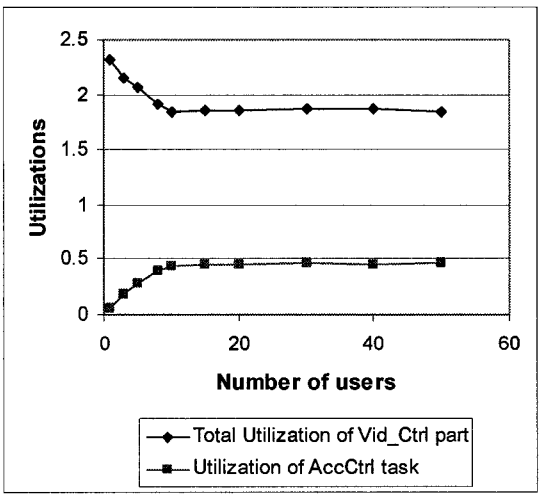


Figure 5.29 The Utilizations of the user part and video part in Experiment 3-3

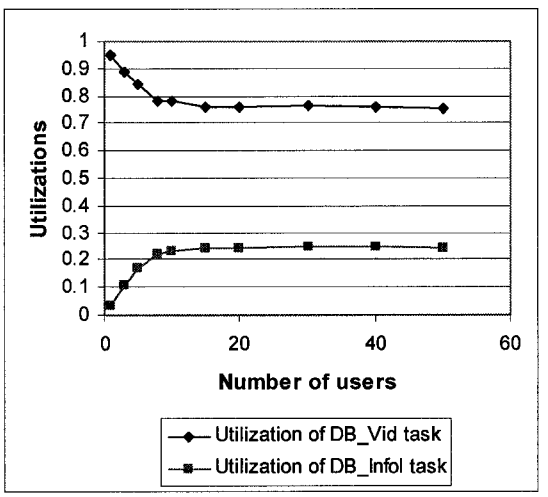


Figure 5.30 Utilizations of DB_Vid and DB_Info in Experiment 3-3

5.6 Conclusions

The case study includes three sets of experiments. Pre-Experiment is to get the proper capacities of tasks in order to remove software bottlenecks. The second set of experiments is the application of the CFS scheduler to the processor ApplicCPU. The third set of experiments applyies CFS scheduling to DB_CPU. Based on these experiments, the following conclusions are reached.

1. CFS scheduling can prevent the starvation of the video part.

2. CFS scheduling does not have to be used at all the processing nodes in a system to have the desired effect. It should be done at the lowest shared resource in the system.

3. The interactions between tasks in different layers affect guarantee shares.

4. Cap shares can strictly limit group utilization. If the tasks of a computer system block each other, cap share is better than the guarantee share, so that other tasks in other groups aren't starved due to blocking.

5. In general, tasks in the groups with small group shares are slowed down. However, for the tasks in upper layers, their response times may not be controlled by their group shares.

6. It is essential to assign a large group share to the group which contains a task in a lower layer and with large service demand.

Chapter 6 Conclusions

The first goal of this thesis is to apply the CFS scheduling policy to the LQN model, which was described in Chapter 3. A new component element 'group' was created in the LQN Meta model. Some extensions were made to the LQN XML schema and the LQN graphical notations to support the CFS scheduling.

The main goal of this research was to develop the PARASOL CFS scheduler with three features: basic CFS scheduling, group CFS scheduling and mixed CFS scheduling. The granularity of the previous PARASOL scheduler was a PARASOL task. The basic CFS scheduler works on the task level and is fair to tasks, and is similar somewhat to the existing round robin scheduler. However, tasks are guaranteed a specific share of the CPU, unlike the default scheduler. The next scheduling type, group scheduling, introduces the concept of groups of tasks which are allocated CPU time rather than at the individual task level. Finally, the mixed CFS scheduler introduces the concepts of "guarantee" and "cap" to the basic and group schedulers. The cap scheduler enforces an upper bound on the group's share whereas the guarantee scheduler guarantees a lower bound. The mixed scheduler permits both cap and guarantee on the same CPU at the same time. Group shares are ignored if the processor is not running the CFS scheduler. Task priorities are ignored if the converse is true. The tests in Chapter 4 demonstrate that the CFS scheduler functions correctly.

Extensions are made to the LQN simulator, Lqsim, to incorporate the CFS scheduling policy. All the experiments in the case study are conducted using Lqsim.

In the case study, the performance impacts of the CFS scheduler were explored. Some conclusions were reached. They are:

1. CFS scheduling does not have to be used at all the nodes in a system to have the desired effect. It should be done at the lowest shared resource in the system.

2. Cap shares can strictly limit group utilization. If the tasks of a computer system block each other, using cap share is better than using guarantee share, so that other tasks in other groups are not starved due to blocking. Therefore, cap share is suitable for general systems regardless of whether there are interactions between the tasks in the system or not.

3. If a group contains a task which is in a lower layer in a LQN model and also has a larger service demand, the group deserves a large share. The large share decreases the blocking time of the task and then decreases the blocking time of tasks in upper layers.

4. In general, using the group CFS scheduler, the response times of tasks in groups are inversely related to their group shares. As a consequence of conclusion 3, the response time of the tasks in upper layers may not be controlled by their shares.

The future work of the thesis may be:

1. Make extensions to the LQN analytical solver, lqns, to support the CFS scheduler. In [32], an analytic model was developed where each class in a conventional queueing network was given a guarantee share. This could serve as a starting point.

2. Find a more efficient way to update the group-rqs after each group's execution, and then reduce the computation overhead.

References

[1] Attahiru Sule Alfa, Yu-Fei Shi, "A discrete time-limited vacation model for the fair share scheduler", *Telecommunication Systems*, Vol. 13, pp167-197, 2000.

[2] Andrea C. Arpaci-Dusseau, David E. Culler, "Extending Proportional-Share Scheduling to a Network of Workstations", *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, USA, Jul, 1997.

[3] Jerry Banks, John S. Carson II, Barry L. Nelson, "Discrete-event system simulation", Upper Saddle River, N.J. Prentice Hall, c.1996, 2nd edition.

[4] Ethan Bolker, "Fair Share Modeling for Large Systems: Aggregation, Hierarchical Decomposition and Randomization", *CMG-CONFERENCE*, pp 807-818, Anaheim, CA, USA, Dec, 2002.

[5] Ethan Bolker, Yiping Ding, "On the Performance Impact of Fair Share Scheduling", *CMG Conf*, pp71-81, Orlando, FL, USA, Dec, 2000.

[6] Philip Carcia, Peter Shankar, Michael Connolly, "Fair-share scheduling in the Linux kernel", Online: http://www.cse.lehigh.edu/~pcg2/fairshare.pdf, [retrieved May, 2008].

[7] Abhishek Chandra, Micah Adler, Pawan Goyal, Prashant Shenoy, "Surplus Fair Scheduling: A Proportional-Share CPU Scheduling Algorithm for Symmetric Multiprocessors", *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, Vol. 4, San Diego, California, 2000.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Chapter 13: Red-Black Trees", *Introduction to Algorithms*, Second Edition, pp. 273–301, MIT Press and McGraw-Hill, 2001.

[9] Erik Elmroth, Peter Gardfjall, "Design and Evaluation of a Decentralized System for Grid-wide Fairshare Scheduling", *First International Conference on e-Science and Grid Computing,* issue, 5-8, Umea, Sweden, Dec. 2005.

[10] Greg Franks, Aiex Hubbard, Shikharesh Majumdar, Dorina Petriu, Jerome Rolia, Murray Woodside, "A toolset for Performance Engineering and Software Design of Client-Server Systems", *Performance Evaluation*, Vol. 24, pp117-135, Nov, 1995.

[11] Greg Franks, Dorina Petriu, Murray Woodside, Jing Xu, Peter Tregunno, "Layered Bottlenecks and Their Mitigation", *Proceedings of the Third International Conference on the Quantative Evaluation of Systems (QEST)*, pp11-14, Riverside, CA, USA, Sep, 2006.

[12] Greg Franks, Peter Maly, Murray Woodside, Dorina C. Petriu, Alex Hubbard, "Layered Queueing Network Solver and Simulator User Manual. Revision: 6840", Department of Systems and Computer Engineering, Carleton University, Dec, 2005.

[13] Greg Franks, Tariq Al-Omari, Murray Woodside, Olivia Das, Salem Derisavi, "Enhanced Modeling and Solution of Layered Queueing Networks", *IEEE Transactions on Software Engineering*, Aug. 2008.

[14] M. Tim Jones, "Inside the Linux scheduler", IBM. June 2006, online: http://www.ibm.com/developerworks/linux/library/l-scheduler/, [retrieved Aug, 2007].

[15] Salil S. Kanhere, Harish Sethu, Apha B. Parekh, "Fair and Efficient Packet Scheduling Using Elastic Round Robin", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 13, pp324 – 336, March, 2002.

[16] J. Kay, P. Lauder, "A Fair Share Scheduler", *Communications of the ACM*, Vol. 31, pp. 44-55, 1988.

[17] Stephen D. Kleban, Scott H. Clearwater. "Fair Share on High Performance Computing Systems: What Does Fair Really Mean?", *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID.03),* Tokyo, Japan, May, 2003.

[18] Avinesh Kumar, "Multiprocessing with the Completely Fair Scheduler: Introducing the CFS for Linux", Jan 2008, Online: http://www.ibm.com/developerworks/linux/library/l-cfs/index.html [retrieved Feb, 2008].

[19] Tom Laramee, "Tutorial On Event Driven Simulations of Network Protocols", December 1995, online: http://www.winslam.com/laramee/sim/index.html, [retrieved Sep, 2008].

[20] John Nagle, "On packet switches with infinite storage", *IEEE Transactions on Communications*, 35(4):435–438, April 1987.

[21] John E. Neilson. "PARASOL Users Manual (Version 3.1)", School of Computer Science, Carleton University, Ottawa, Ontario, Canada.

[22] John E. Neilson, "PARASOL: A simulator for distributed and/or parallel systems." *Technical Report SCS TR-192*, School of Computer Science, Carleton University, Ottawa, Ontario, Canada, May 1991.

[23] A.K. Parekh, "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks", PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992.

[24] David Petrou, John W. Milford, Garth A. Gibson, "Implementing Lottery Scheduling: Matching the Specializations in Traditional Schedulers", *Proceedings of the 1999 USENIX Annual Technical Conference,* June 6–11, 1999.

[25] Stewart Robinson, "Simulation - The practice of model development and use", Wiley, 2004.

[26] Rolia, J. A. , Sevcik, K. A., "The Method of Layers", *IEEE Transactions on Software Engineering,* Vol. 21-8, pp. 689-700, Aug 1995.

[27] Dimitrios Stiliadis, Anujan Varma, "Latency-rate servers: a general model for analysis of traffic scheduling algorithms", *IEEE/ACM Transactions on Networking*, pp. 611–624, IEEE Press Piscataway, NJ, USA, 1998.

[28] Carl A. Waldspurger, "Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management", PhD thesis, Massachusetts Institute of Technology, September 1995.

[29] Carl A. Waldspurger, William E. Weihl, "Lottery Scheduling: Flexible Proportional-Share Resource Management", *In Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI),* pp1–11, Nov, 1994.

[30] Qi Allen Wang, Chung-Horng Lung, Shikharesh Majumdar, "Effective Fair Share Resource Management Algorithms in Support of Quality-of-Service", *Proc. of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, Philadelphia, PA, July 2005.

[31] Roger Whitney, "CS 660: Red-Black tree notes". San Diego State University, Online: http://www.eli.sdsu.edu/courses/fall95/cs660/notes/RedBlackTree/RedBlack.html [retrieved Oct, 2007].

[32] Murray Woodside, "Controllability of computer performance tradeoffs obtained using controlled-share queue schedulers", *IEEE Transactions on Software Engineering*, Vol. 12, issue 10, pp1041 – 1048, 1986.

[33] Murray Woodside, "Tutorial on LQN model ",online: http://www.sce.carleton.ca/ courses/ sysc-5101/w08/notes6b.pdf, [retrieved Oct, 2008].

[34] Murray Woodside, Greg Franks, Dorina Petriu, "The Future of Software Performance Engineering", *Proceedings of the Future of Software Engineering,* Minneapolis, MN, May 23-25, 2007.

[35] Murray Woodside, John E. Neilson, Shikharesh Majumdar, "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-Like Distributed Software", *IEEE Transactions on Computers*, Vol. 44, pp. 20-34, Aug. 1995.

[36] Xiuping Wu, "An Approach to Predicting Performance for Component Based Systems", Thesis for the degree of Master of Applied Science, Carleton University, July 2003.

[37] Jing Xu, Greg. Franks, Shikharesh Majumdar, John Neilson, Dorina Petriu, Jerome Rolia, Murray Woodside, "Performance Analysis of Distributed Server Systems", *In Proceedings of the 6th International Conference on Software Quality*, pp15-26, Ottawa, Canada, October 1996.

[38] Altova XMLSpy 2008 Professional Edition, online: http://www.altova.com/ manual2008/XMLSpy/spyprofessional [retrieved Sep, 2008].

[39] "Solaris 9 8/03 System Administrator Collection, System Administration Guide: Resource Management and Network Services Chapter 9. Fair Share Scheduler". Online:

http://docs.sun.com/app/docs/doc/817-0204/6mg168brv?l=ko&a=view, [retrieved Jan, 2008].

[40] Online: http://lxr.linux.no/linux+v2.6.23.17/Documentation/sched-design-CFS.txt [retrieved Dec, 2007].

[41] Trees II: red-black trees, Online: http://lwn.net/Articles/184495/, [retrieved Feb, 2008].

Appendix A: LQN core schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2006 rel. 3 sp1 (http://www.altova.com) by Greg Franks (Carle) -->
<!--
   Copyright March 2004, the Real-Time and Distributed Systems Group,
   Department of Systems and Computer Engineering,
   Carleton University, Ottawa, Ontario, Canada. K1S 5B6
-->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <!--
      lqn-core is the kernel of lqn sub-model and assembly model
-->
      <xsd:element name="lqn-core" type="Lqn-CoreType"/>
      <xsd:complexType name="Lqn-CoreType">
            <xsd:sequence>
                  <xsd:element name="processor" type="ProcessorType" maxOccurs="unbounded"/>
                  <xsd:element name="slot" type="SlotType" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
      </xsd:complexType>
      <!--
       LQNCoreVersionType
-->
      <xsd:attributeGroup name="LQNCoreVersion">
            <xsd:attribute name="lqncore-schema-version" type="xsd:decimal" fixed="1.0"/>
      </xsd:attributeGroup>
      <!--
       SlotType
 -->
      <xsd:complexType name="SlotType">
            <xsd:sequence>
                  <xsd:element name="Interface">
                        <xsd:complexType>
                              <xsd:sequence>
```

```
                                    <xsd:element name="in-port" minOccurs="0" maxOccurs="unbounded">
                                          <xsd:complexType>
                                                <xsd:attribute name="name" type="xsd:string"
use="required"/>
                                                <xsd:attribute name="connect-from">
                                                      <xsd:simpleType>
                                                            <xsd:list itemType="xsd:string"/>
                                                      </xsd:simpleType>
                                                </xsd:attribute>
                                                <xsd:attribute name="description" type="xsd:string"/>
                                          </xsd:complexType>
                                    </xsd:element>
                                    <xsd:element name="out-port" minOccurs="0" maxOccurs="unbounded">
                                          <xsd:complexType>
                                                <xsd:attribute name="name" type="xsd:string"
use="required"/>
                                                <xsd:attribute name="connect-to">
                                                      <xsd:simpleType>
                                                            <xsd:list itemType="xsd:string"/>
                                                      </xsd:simpleType>
                                                </xsd:attribute>
                                                <xsd:attribute name="description" type="xsd:string"/>
                                          </xsd:complexType>
                                    </xsd:element>
                              </xsd:sequence>
                        </xsd:complexType>
                  </xsd:element>
                  <xsd:element name="binding" type="BindType" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="bind-target" type="xsd:string" use="required"/>
            <xsd:attribute name="id" type="xsd:string" use="required"/>
            <xsd:attribute name="replic_num" type="xsd:int"/>
      </xsd:complexType>
      <!--
       SlotType definition
-->
```

```
	<xsd:complexType name="BindType">
		<xsd:sequence>
			<xsd:element name="parameter" minOccurs="0" maxOccurs="unbounded">
				<xsd:complexType>
					<xsd:attribute name="name" type="xsd:string" use="required"/>
					<xsd:attribute name="value" type="xsd:string" use="required"/>
				</xsd:complexType>
			</xsd:element>
			<xsd:element name="processor-binding" minOccurs="0" maxOccurs="unbounded">
				<xsd:complexType>
					<xsd:attribute name="source" type="xsd:string" use="required"/>
					<xsd:attribute name="target" type="xsd:string" use="required"/>
				</xsd:complexType>
			</xsd:element>
			<xsd:element name="port-binding" minOccurs="0" maxOccurs="unbounded">
				<xsd:complexType>
					<xsd:attribute name="source" type="xsd:string" use="required"/>
					<xsd:attribute name="target" type="xsd:string" use="required"/>
				</xsd:complexType>
			</xsd:element>
		</xsd:sequence>
	</xsd:complexType>
	<!--
	 ProcessorType
-->
	<xsd:complexType name="ProcessorType">
		<xsd:annotation>
			<xsd:documentation>Processors run tasks.</xsd:documentation>
		</xsd:annotation>
		<xsd:sequence>
			<xsd:element name="result-processor" type="OutputResultType" minOccurs="0"
maxOccurs="unbounded"/>
			<xsd:choice>
				<xsd:element name="group" type="GroupType" maxOccurs="unbounded"/>
				<xsd:element name="task" type="TaskType" maxOccurs="unbounded"/>
			</xsd:choice>
```

```
            </xsd:sequence>
            <xsd:attribute name="multiplicity" type="xsd:nonNegativeInteger" default="1"/>
            <xsd:attribute name="speed-factor" type="SrvnFloat" default="1"/>
            <xsd:attribute name="scheduling" type="SchedulingType" default="fcfs"/>
            <xsd:attribute name="replication" type="xsd:nonNegativeInteger" default="1"/>
            <xsd:attribute name="quantum" type="SrvnFloat" default="0"/>
            <xsd:attribute name="name" type="xsd:string" use="required"/>
      </xsd:complexType>
      <!--
      SchedulingType for processors.
-->
      <xsd:simpleType name="SchedulingType">
            <xsd:restriction base="xsd:string">
                  <xsd:enumeration value="fcfs"/>
                  <xsd:enumeration value="ps"/>
                  <xsd:enumeration value="pp"/>
                  <xsd:enumeration value="inf"/>
                  <xsd:enumeration value="rand"/>
                  <xsd:enumeration value="hol"/>
                  <xsd:enumeration value="ps-hol"/>
                  <xsd:enumeration value="ps-pp"/>
                  <xsd:enumeration value="cfs"/>
            </xsd:restriction>
      </xsd:simpleType>
      <!--
      GroupType
-->
      <xsd:complexType name="GroupType">
            <xsd:sequence>
                  <xsd:element name="result-group" type="OutputResultType" minOccurs="0"
maxOccurs="unbounded"/>
                  <xsd:element name="task" type="TaskType" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:string" use="required"/>
            <xsd:attribute name="capflag" type="xsd:string" use="optional"/>
            <xsd:attribute name="share" type="xsd:double"/>
```

```
      </xsd:complexType>
      <!--
      TaskSchedulingType
-->
      <xsd:simpleType name="TaskSchedulingType">
            <xsd:restriction base="xsd:string">
                  <xsd:enumeration value="ref"/>
                  <xsd:enumeration value="fcfs"/>
                  <xsd:enumeration value="pri"/>
                  <xsd:enumeration value="hol"/>
                  <xsd:enumeration value="burst"/>
                  <xsd:enumeration value="poll"/>
                  <xsd:enumeration value="inf"/>
                  <xsd:enumeration value="semaphore"/>
                  <xsd:enumeration value="cfs"/>
            </xsd:restriction>
      </xsd:simpleType>
      <!--
      TaskType
-->
      <xsd:complexType name="TaskType">
            <xsd:sequence>
                  <xsd:element name="result-task" type="OutputResultType" minOccurs="0"
maxOccurs="unbounded"/>
                  <xsd:element name="entry" type="EntryType" maxOccurs="unbounded"/>
                  <xsd:element name="service" type="ServiceType" minOccurs="0" maxOccurs="unbounded"/>
                  <xsd:element name="task-activities" type="TaskActivityGraph" minOccurs="0">
                        <xsd:keyref name="MatchingTaskActivities" refer="ActivityDefs">
                              <xsd:selector xpath=".//activity"/>
                              <xsd:field xpath="@name"/>
                        </xsd:keyref>
                        <xsd:keyref name="MatchingLoopHeadActivities" refer="ActivityDefs">
                              <xsd:selector xpath="./precedence/post-LOOP"/>
                              <xsd:field xpath="@head"/>
                        </xsd:keyref>
                        <xsd:keyref name="MatchingLoopEndActivities" refer="ActivityDefs">
```

```
                        <xsd:selector xpath="./precedence/post-LOOP"/>
                        <xsd:field xpath="@end"/>
                    </xsd:keyref>
                    <xsd:keyref name="MatchingReplyActivities" refer="ActivityDefs">
                        <xsd:selector xpath="./reply-entry/reply-activity"/>
                        <xsd:field xpath="@name"/>
                    </xsd:keyref>
                    <xsd:key name="ActivityDefs">
                        <xsd:selector xpath="./activity"/>
                        <xsd:field xpath="@name"/>
                    </xsd:key>
                </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:string" use="required"/>
            <xsd:attribute name="multiplicity" type="xsd:nonNegativeInteger" default="1"/>
            <xsd:attribute name="replication" type="xsd:nonNegativeInteger" default="1"/>
            <xsd:attribute name="scheduling" type="TaskSchedulingType" default="fcfs"/>
            <xsd:attribute name="think-time" type="SrvnFloat" default="0"/>
            <xsd:attribute name="priority" type="xsd:nonNegativeInteger" default="0"/>
            <xsd:attribute name="queue-length" type="xsd:nonNegativeInteger" default="0"/>
            <xsd:attribute name="activity-graph" type="TaskOptionType"/>
        </xsd:complexType>
        <!--
        TaskOptionType
-->
        <xsd:simpleType name="TaskOptionType">
            <xsd:restriction base="xsd:string">
                <xsd:enumeration value="YES"/>
                <xsd:enumeration value="NO"/>
            </xsd:restriction>
        </xsd:simpleType>
        <!--
        EntryType
-->
        <xsd:complexType name="EntryType">
            <xsd:sequence>
```

```
                <xsd:element name="result-entry" type="OutputResultType" minOccurs="0"
maxOccurs="unbounded"/>
                <xsd:element name="service-time-distribution" type="OutputEntryDistributionType"
minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="forwarding" type="EntryMakingCallType" minOccurs="0"
maxOccurs="unbounded"/>
                <xsd:choice>
                     <xsd:element name="entry-activity-graph" type="EntryActivityGraph"
minOccurs="0"/>
                     <xsd:element name="entry-phase-activities" type="PhaseActivities"
minOccurs="0">
                          <xsd:unique name="UniquePhaseNumber">
                               <xsd:selector xpath="activity"/>
                               <xsd:field xpath="@phase"/>
                          </xsd:unique>
                     </xsd:element>
                </xsd:choice>
           </xsd:sequence>
           <xsd:attribute name="type" use="required">
                <xsd:simpleType>
                     <xsd:restriction base="xsd:string">
                          <xsd:enumeration value="PH1PH2"/>
                          <xsd:enumeration value="GRAPH"/>
                          <xsd:enumeration value="NONE"/>
                     </xsd:restriction>
                </xsd:simpleType>
           </xsd:attribute>
           <xsd:attribute name="name" type="xsd:string" use="required"/>
           <xsd:attribute name="open-arrival-rate" type="SrvnFloat"/>
           <xsd:attribute name="priority" type="xsd:int"/>
           <xsd:attribute name="semaphore" type="SemaphoreType"/>
     </xsd:complexType>
     <!--
     ActivityGraphBase
-->
     <xsd:complexType name="ActivityGraphBase">
```

```
            <xsd:sequence>
                  <xsd:element name="activity" type="ActivityDefType" maxOccurs="unbounded"/>
                  <xsd:element name="precedence" type="PrecedenceType" minOccurs="0"
maxOccurs="unbounded"/>
            </xsd:sequence>
      </xsd:complexType>
      <!--
      ActivityDefBase
-->
      <xsd:complexType name="ActivityDefBase">
            <xsd:sequence>
                  <xsd:element name="result-join-delay" type="OutputResultForwardingANDJoinDelay"
minOccurs="0" maxOccurs="unbounded"/>
                  <xsd:element name="result-forwarding" type="OutputResultForwardingANDJoinDelay"
minOccurs="0" maxOccurs="unbounded"/>
                  <xsd:element name="service-time-distribution" type="OutputDistributionType"
minOccurs="0" maxOccurs="unbounded"/>
                  <xsd:element name="result-activity" type="OutputResultType" minOccurs="0"
maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:string" use="required"/>
            <xsd:attribute name="host-demand-mean" type="SrvnFloat" use="required"/>
            <xsd:attribute name="host-demand-cvsq" type="SrvnFloat"/>
            <xsd:attribute name="think-time" type="SrvnFloat"/>
            <xsd:attribute name="max-service-time" type="SrvnFloat"/>
            <xsd:attribute name="call-order" type="CallOrderType" default="STOCHASTIC"/>
      </xsd:complexType>
      <!--
      ActivityDefType (activities for task activity graphs)
-->
      <xsd:complexType name="ActivityDefType">
            <xsd:complexContent>
                  <xsd:extension base="ActivityDefBase">
                        <xsd:sequence>
                              <xsd:choice>
                                    <xsd:group ref="Call-List-Group"/>
```

```
                                    <xsd:group ref="Activity-CallGroup" maxOccurs="unbounded"/>
                              </xsd:choice>
                        </xsd:sequence>
                        <xsd:attribute name="bound-to-entry" type="xsd:string"/>
                  </xsd:extension>
            </xsd:complexContent>
      </xsd:complexType>
      <!--
      EntryActivityDefType (activities for entry activity graphs)

      *** NOTE: this is not supported yet by the parser.
      *** NOTE: first-activity can equal anything, its very presence means everything
-->
      <xsd:complexType name="EntryActivityDefType">
            <xsd:complexContent>
                  <xsd:extension base="ActivityDefBase">
                        <xsd:sequence>
                              <xsd:choice>
                                    <xsd:group ref="Call-List-Group"/>
                                    <xsd:group ref="Activity-CallGroup" maxOccurs="unbounded"/>
                              </xsd:choice>
                        </xsd:sequence>
                        <xsd:attribute name="first-activity" type="xsd:string"/>
                  </xsd:extension>
            </xsd:complexContent>
      </xsd:complexType>
      <!--
      ActivityPhasesType (activities for specifying phases)
-->
      <xsd:complexType name="ActivityPhasesType">
            <xsd:complexContent>
                  <xsd:extension base="ActivityDefBase">
                        <xsd:sequence>
                              <xsd:choice>
                                    <xsd:group ref="Call-List-Group"/>
                                    <xsd:group ref="Activity-CallGroup" maxOccurs="unbounded"/>
```

```
                        </xsd:choice>
                    </xsd:sequence>
                    <xsd:attribute name="phase" use="required">
                        <xsd:simpleType>
                            <xsd:restriction base="xsd:positiveInteger">
                                <xsd:minInclusive value="1"/>
                                <xsd:maxInclusive value="3"/>
                            </xsd:restriction>
                        </xsd:simpleType>
                    </xsd:attribute>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
        <!--
        PrecedenceType
-->
        <xsd:complexType name="PrecedenceType">
            <xsd:sequence>
                <xsd:choice>
                    <xsd:element name="pre" type="SingleActivityListType"/>
                    <xsd:element name="pre-OR" type="ActivityListType"/>
                    <xsd:element name="pre-AND" type="AndJoinListType"/>
                </xsd:choice>
                <xsd:choice>
                    <xsd:element name="post" type="SingleActivityListType" minOccurs="0"/>
                    <xsd:element name="post-OR" type="OrListType" minOccurs="0"/>
                    <xsd:element name="post-AND" type="ActivityListType" minOccurs="0"/>
                    <xsd:element name="post-LOOP" type="ActivityLoopListType" minOccurs="0"/>
                </xsd:choice>
            </xsd:sequence>
        </xsd:complexType>
        <!--
           ActivityType (activity in <precedence> relationship)
-->
        <xsd:complexType name="ActivityType">
            <xsd:attribute name="name" type="xsd:string" use="required"/>
```

```
      </xsd:complexType>
      <!--
      ActivityLoopType (for generalized loops in <precendence> relationship)
-->
      <xsd:complexType name="ActivityLoopType">
            <xsd:complexContent>
                  <xsd:extension base="ActivityType">
                        <xsd:attribute name="count" type="SrvnFloat" use="required"/>
                  </xsd:extension>
            </xsd:complexContent>
      </xsd:complexType>
      <!--
      ActivityOrType (for generalized loops in <precendence> relationship)
-->
      <xsd:complexType name="ActivityOrType">
            <xsd:complexContent>
                  <xsd:extension base="ActivityType">
                        <xsd:attribute name="prob" type="xsd:string" default="1"/>
                  </xsd:extension>
            </xsd:complexContent>
      </xsd:complexType>
      <!--
      SingleActivityListType
-->
      <xsd:complexType name="SingleActivityListType">
            <xsd:sequence>
                  <xsd:element name="activity" type="ActivityType"/>
            </xsd:sequence>
      </xsd:complexType>
      <!--
      OrListType
-->
      <xsd:complexType name="OrListType">
            <xsd:sequence>
                  <xsd:element name="activity" type="ActivityOrType" maxOccurs="unbounded"/>
            </xsd:sequence>
```

```
      </xsd:complexType>
      <!--
      ActivityListType
-->
      <xsd:complexType name="ActivityListType">
            <xsd:sequence>
                  <xsd:element name="activity" type="ActivityType" maxOccurs="unbounded"/>
            </xsd:sequence>
      </xsd:complexType>
      <!--
      AndJoinListType
-->
      <xsd:complexType name="AndJoinListType">
            <xsd:sequence>
                  <xsd:element name="activity" type="ActivityType" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="quorum" type="xsd:nonNegativeInteger" default="0"/>
      </xsd:complexType>
      <!--
      ActivityLoopListType
-->
      <xsd:complexType name="ActivityLoopListType">
            <xsd:sequence>
                  <xsd:element name="activity" type="ActivityLoopType" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="end" type="xsd:string"/>
      </xsd:complexType>
      <!--
      TaskActivityGraph
-->
      <xsd:complexType name="TaskActivityGraph">
            <xsd:complexContent>
                  <xsd:extension base="ActivityGraphBase">
                        <xsd:sequence>
                              <xsd:element name="reply-entry" minOccurs="0" maxOccurs="unbounded">
                                    <xsd:complexType>
```

```
                                    <xsd:group ref="ReplyActivity"/>
                                    <xsd:attribute name="name" type="xsd:string" use="required"/>
                              </xsd:complexType>
                        </xsd:element>
                  </xsd:sequence>
            </xsd:extension>
      </xsd:complexContent>
</xsd:complexType>
<!--
EntryActivityGraph
-->
<xsd:complexType name="EntryActivityGraph">
      <xsd:complexContent>
            <xsd:extension base="ActivityGraphBase">
                  <xsd:group ref="ReplyActivity"/>
            </xsd:extension>
      </xsd:complexContent>
</xsd:complexType>
<!--
PhaseActivities
-->
<xsd:complexType name="PhaseActivities">
      <xsd:sequence>
            <xsd:element name="activity" type="ActivityPhasesType" maxOccurs="3"/>
      </xsd:sequence>
</xsd:complexType>
<!--
ReplyActivity
-->
<xsd:group name="ReplyActivity">
      <xsd:sequence>
            <xsd:element name="reply-activity" maxOccurs="unbounded">
                  <xsd:complexType>
                        <xsd:attribute name="name" type="xsd:string" use="required"/>
                  </xsd:complexType>
            </xsd:element>
```

```
            </xsd:sequence>
      </xsd:group>
      <!--
      ServiceType
-->
      <xsd:complexType name="ServiceType">
            <xsd:attribute name="name" type="xsd:string" use="required"/>
      </xsd:complexType>
      <!--
      CallOrderType
-->
      <xsd:simpleType name="CallOrderType">
            <xsd:restriction base="xsd:string">
                  <xsd:enumeration value="STOCHASTIC"/>
                  <xsd:enumeration value="DETERMINISTIC"/>
                  <xsd:enumeration value="LIST"/>
            </xsd:restriction>
      </xsd:simpleType>
      <!--
      Activity-CallGroup
-->
      <xsd:group name="Activity-CallGroup">
            <xsd:sequence>
                  <xsd:choice>
                        <xsd:element name="synch-call" type="ActivityMakingCallType" minOccurs="0"/>
                        <xsd:element name="asynch-call" type="ActivityMakingCallType" minOccurs="0"/>
                  </xsd:choice>
            </xsd:sequence>
      </xsd:group>
      <!--
      MakingCallType
-->
      <xsd:complexType name="MakingCallType">
            <xsd:sequence>
                  <xsd:element name="result-call" type="OutputResultType" minOccurs="0"
maxOccurs="unbounded"/>
```

```
            </xsd:sequence>
            <xsd:attribute name="dest" type="xsd:string" use="required"/>
            <xsd:attribute name="fanout" type="xsd:int"/>
            <xsd:attribute name="fanin" type="xsd:int"/>
      </xsd:complexType>
      <!--
      ActivityMakingCallType
-->
      <xsd:complexType name="ActivityMakingCallType">
            <xsd:complexContent>
                  <xsd:extension base="MakingCallType">
                        <xsd:attribute name="calls-mean" type="SrvnFloat" use="required"/>
                  </xsd:extension>
            </xsd:complexContent>
      </xsd:complexType>
      <!--
      EntryMakingCallType
-->
      <xsd:complexType name="EntryMakingCallType">
            <xsd:complexContent>
                  <xsd:extension base="MakingCallType">
                        <xsd:attribute name="prob" type="SrvnFloat" use="required"/>
                  </xsd:extension>
            </xsd:complexContent>
      </xsd:complexType>
      <!--
      Call-List-Group  *** NOT SUPPORTED ***
-->
      <xsd:group name="Call-List-Group">
            <xsd:sequence>
                  <xsd:element name="call-list" minOccurs="0" maxOccurs="unbounded">
                        <xsd:complexType>
                              <xsd:sequence>
                                    <xsd:element name="synch-call" minOccurs="0" maxOccurs="unbounded">
                                          <xsd:complexType>
                                                <xsd:attributeGroup ref="CallListType"/>
```

```
                                    </xsd:complexType>
                                </xsd:element>
                                <xsd:element name="asynch-call" minOccurs="0" maxOccurs="unbounded">
                                    <xsd:complexType>
                                        <xsd:attributeGroup ref="CallListType"/>
                                    </xsd:complexType>
                                </xsd:element>
                            </xsd:sequence>
                        </xsd:complexType>
                    </xsd:element>
                </xsd:sequence>
            </xsd:group>
            <!--
            CallListType   *** NOT SUPPORTED ***
        -->
            <xsd:attributeGroup name="CallListType">
                <xsd:attribute name="dest" type="xsd:string" use="required"/>
                <xsd:attribute name="fanout" type="xsd:int"/>
                <xsd:attribute name="fanin" type="xsd:int"/>
            </xsd:attributeGroup>
            <!--
            SciNotation type
        -->
            <xsd:simpleType name="SciNotation">
                <xsd:restriction base="xsd:string">
                    <xsd:pattern value="[\+|\-]?\d+((\.\d+)?[e|E][\+|\-]\d+)?"/>
                </xsd:restriction>
            </xsd:simpleType>
            <!--
            SRVN float type which is union of decimal and scientific notation
        -->
            <xsd:simpleType name="SrvnFloat">
                <xsd:union memberTypes="SciNotation xsd:decimal"/>
            </xsd:simpleType>
            <!--
            SemaphoreType
```

```
-->
      <xsd:simpleType name="SemaphoreType">
            <xsd:restriction base="xsd:string">
                  <xsd:enumeration value="signal"/>
                  <xsd:enumeration value="wait"/>
            </xsd:restriction>
      </xsd:simpleType>
      <!--
      OutputResultType
-->
      <xsd:complexType name="OutputResultType">
            <xsd:sequence>
                  <xsd:element name="result-conf-95" minOccurs="0" maxOccurs="unbounded">
                        <xsd:complexType>
                              <xsd:attributeGroup ref="ResultContentType"/>
                        </xsd:complexType>
                  </xsd:element>
                  <xsd:element name="result-conf-99" minOccurs="0" maxOccurs="unbounded">
                        <xsd:complexType>
                              <xsd:attributeGroup ref="ResultContentType"/>
                        </xsd:complexType>
                  </xsd:element>
            </xsd:sequence>
            <xsd:attributeGroup ref="ResultContentType"/>
      </xsd:complexType>
      <!--
      OutputResultForwardingANDJoinDelay
-->
      <xsd:complexType name="OutputResultForwardingANDJoinDelay">
            <xsd:sequence>
                  <xsd:element name="result-conf-95" minOccurs="0" maxOccurs="unbounded">
                        <xsd:complexType>
                              <xsd:attribute name="waiting" type="SrvnFloat" use="optional"/>
                              <xsd:attribute name="join-waiting" type="SrvnFloat" use="optional"/>
                              <xsd:attribute name="join-variance" type="SrvnFloat" use="optional"/>
                        </xsd:complexType>
```

```
                </xsd:element>
                <xsd:element name="result-conf-99" minOccurs="0" maxOccurs="unbounded">
                    <xsd:complexType>
                        <xsd:attribute name="waiting" type="SrvnFloat" use="optional"/>
                        <xsd:attribute name="join-waiting" type="SrvnFloat" use="optional"/>
                        <xsd:attribute name="join-variance" type="SrvnFloat" use="optional"/>
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="dest" type="xsd:string" use="required"/>
            <xsd:attribute name="waiting" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="join-waiting" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="join-variance" type="SrvnFloat" use="optional"/>
        </xsd:complexType>
        <!--
        ResultContentType attribute group
-->
        <xsd:attributeGroup name="ResultContentType">
            <xsd:attribute name="utilization" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="throughput" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="phase1-utilization" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="phase2-utilization" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="phase3-utilization" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="throughput-bound" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="proc-utilization" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="proc-waiting" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="phase1-proc-waiting" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="phase2-proc-waiting" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="phase3-proc-waiting" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="squared-coeff-variation" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="open-wait-time" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="service-time" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="phase1-service-time" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="phase2-service-time" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="phase3-service-time" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="service-time-variance" type="SrvnFloat" use="optional"/>
```

```
            <xsd:attribute name="phase1-service-time-variance" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="phase2-service-time-variance" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="phase3-service-time-variance" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="prob-exceed-max-service-time" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="waiting" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="waiting-variance" type="SrvnFloat" use="optional"/>
      </xsd:attributeGroup>
      <!--
      Axis type.
-->
      <xsd:simpleType name="AxisType">
            <xsd:restriction base="xsd:string">
                  <xsd:enumeration value="linear"/>
                  <xsd:enumeration value="log"/>
            </xsd:restriction>
      </xsd:simpleType>
      <!--
      OutputDistributionType
-->
      <xsd:complexType name="OutputDistributionType">
            <xsd:sequence>
                  <xsd:element name="underflow-bin" type="HistogramBinType" minOccurs="0"/>
                  <xsd:element name="histogram-bin" type="HistogramBinType" minOccurs="0"
maxOccurs="unbounded"/>
                  <xsd:element name="overflow-bin" type="HistogramBinType" minOccurs="0"/>
            </xsd:sequence>
            <xsd:attribute name="min" type="SrvnFloat" use="required"/>
            <xsd:attribute name="mid-point" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="max" type="SrvnFloat" use="required"/>
            <xsd:attribute name="bin-size" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="number-bins" type="xsd:nonNegativeInteger" default="20"/>
            <xsd:attribute name="x-samples" type="AxisType" default="linear"/>
            <xsd:attribute name="mean" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="std-dev" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="skew" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="kurtosis" type="SrvnFloat" use="optional"/>
```

```
      </xsd:complexType>
      <!--
      OutputEntryDistributionType
-->
      <xsd:complexType name="OutputEntryDistributionType">
            <xsd:complexContent>
                  <xsd:extension base="OutputDistributionType">
                        <xsd:attribute name="phase" type="xsd:nonNegativeInteger" use="required"/>
                  </xsd:extension>
            </xsd:complexContent>
      </xsd:complexType>
      <!--
      HistogramBinType
-->
      <xsd:complexType name="HistogramBinType">
            <xsd:attribute name="begin" type="SrvnFloat" use="required"/>
            <xsd:attribute name="end" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="prob" type="SrvnFloat" use="required"/>
            <xsd:attribute name="conf-95" type="SrvnFloat" use="optional"/>
            <xsd:attribute name="conf-99" type="SrvnFloat" use="optional"/>
      </xsd:complexType>
</xsd:schema>
```

Appendix B: Grammar of the old style LQN input File

1. Input File Grammar

<LQN_input_file> → *<general_info>* *<processor_info>* *<group_info>*$_{opt}$
<task_info> *<entry_info>* {*<activity_info>*}$_0$

2. General information section

<general_info> → G *<comment>* *<conv_val>* *<it_limit>* *<print_int>*$_{opt}$
<underrelax_coeff>$_{opt}$ *<end_list>*
<comment> → *<string>* /* *comment on the model* */
<conv_val> → *<real>* /* *convergence value* */
<it_limit> → *<integer>* /* *max. nb. of iterations* */
<print_int> → *<integer>* /* *intermed. res. print interval* */
<underrelax_coeff> → *<real>* /* *under_relaxation coefficient* */
<end_list> → -1 /* *end_of_list mark* */
<string> → " *<text>* "

3. Processor information section

<processor_info> → P *<np>* *<p_decl_list>*
<np> → *<integer>* /* *total number of processors* */
<p_decl_list> → {*<p_decl>*}$_1^{np}$ *<end_list>*
<p_decl> → p *<proc_id>* *<scheduling_flag>* *<quantum>*$_{opt}$
<multi_server_flag>$_{opt}$ *<replication_flag>*$_{opt}$ *<proc_rate>*$_{opt}$
<proc_id> → *<integer>* | *<identifier>* /* *processor identifier* */
<scheduling_flag> → f /* *First come, first served* */
| h /* *Head Of Line* */
| p /* *Priority, preemeptive* */
| c *<real>* /* *Completely fair scheduling* */
| s *<real>* /* *Processor sharing* */
| i /* *Infinite or delay* */
| r /* *Random* */
<quantum> → *<real>*
<multi_server_flag> → m *<copies>* /* *Number of duplicates* */
| i /* *Infinite server* */
<replication_flag> → r *<copies>* /* *Number of replicas* */
<proc_rate> → R *<ratio>* /* *Relative proc. speed* */
<copies> → *<integer>*
<ratio> → *<real>*

4. Group information section

<group_info> → U <ng> <g_decl_list> <end_list>
<ng> → <integer> /* total number of groups */
<g_decl_list> → $\{$<g_decl>$\}_1^{ng}$ <end_list>
<g_decl> → g <group_id> <group_share> <cap_flag>$_{opt}$ <proc_id>
<group_id> → <identifier>
<group_share> → <real>
<cap_flag> → c

5. Task information section

<task_info> → T <nt> <t_decl_list>
<nt> → <integer> /* total number of tasks */
<t_decl_list> → $\{$<t_decl>$\}_1^{nt}$ <end_list>
<t_decl> → t <task_id> <task_sched_type> <entry_list>
<queue_length>$_{opt}$ <proc_id> <task_pri>$_{opt}$
<think_time_flag>$_{opt}$ <multi_server_flag>$_{opt}$
<replication_flag>$_{opt}$ <group_flag>$_{opt}$
<t_decl> → t <task_id> S <entry_list> <proc_id> <task_pri>$_{opt}$
<multi_server_flag>$_{opt}$ <replication_flag>$_{opt}$
<task_id> → <integer> | <identifier> /* task identifier */
<task_sched_type> → r /* reference task */
| n /* non-reference task */
| h /* head of line */
| f /* FIFO Scheduling */
| i /* Infinite or delay server */
| p /* Polled scheduling at entries */
| b /* Bursty Reference task */
<entry_list> → $\{$<entry_id>$\}_1^{ne\,t}$ <end_list> /* task t has ne_t entries */
<entry_id> → <integer> | <identifier> /* entry identifier */
<queue_length> → q <integer> /* open class queue length */
<task_pri> → <integer> /* task priority, optional */
<group_flag> → g <identifier> /* Group for scheduling */

6. Entry information section

<entry_info> → E <ne> <entry_decl_list>
<ne> → <integer> /* total number of entries */
<entry_decl_list> → $\{$<entry_decl>$\}_1$ <end_list>
/* k = maximum number of phases */
<entry_decl> → a <entry_id> <arrival_rate>
| A <entry_id> <activity_id>
| F <from_entry> <to_entry> <p_forward>
| H <entry_id> <phase> <hist_min> ':' <hist_max>

<hist_bins> <hist_type>
| M <entry_id> {<max_service_time>}$_1^k$ <end_list>
| P <entry_id> /* Signal Semaphore */
| V <entry_id> /* Wait Semaphore */
| Z <entry_id> {<think_time>}$_1^k$ <end_list>
| c <entry_id> {<coeff_of_variation>}$_1^k$ <end_list>
| f <entry_id> {<p< type_flag>}$_1^k$ <end_list>
| i <from_entry> <to_entry> {<fan_in>}$_1^k$ <end_list>
| o <from_entry> <to_entry> {<fan_out>}$_1^k$ <end_list>
| p <entry_id> <entry_priority>
| s <entry_id> {<service_time>}$_1^k$ <end_list>
| y <from_entry> <to_entry> {<rendezvous>}$_1^k$ <end_list>
| z <from_entry> <to_entry> {<send_no_reply>}$_1^k$
<end_list>

<arrival_rate> → <real> /* open arrival rate to entry */
<coeff_of_variation> → <real> /* squared service time coefficient of variation */
<fan_in> → <integer> /* fan in to this entry */
<fan_out> → <integer> /* fan out of this entry */
<from_entry> → <entry_id> /* Source of a message */
<hist_bins> → <integer> /* Number of bins in histogram. */
<hist_max> → <real> /* Median service time. */
<hist_min> → <real> /* Median service time. */
<hist_type> → log | linear | sqrt /* bin type. */
<max_service_ time> → <real> /* Median service time. */
<p_forward> → <real> /* probability of forwarding */
<phase> → 1 | 2 | 3 /* phase of entry */
<ph_type_flag> → 0 /* stochastic phase */
| 1 /* deterministic phase */
<rate> → <real> /* nb. of calls per arrival */
<rendezvous> → <real> /* mean number of RNVs/ph */
<send_no_reply> → <real> /* mean nb.of non-blck.sends/ph */
<service_time> → <real> /* mean phase service time */
<think_time> → <real> /* Think time for phase. */
<to_entry> → <entry_id> /* Destination of a message */

7. Activity information section

<activity_info> → <activity_defn_list> <activity_connections>$_{opt}$ <end_list>
/* Activity definition. */
<activity_defn_list> → {<activity_defn>}$_1^{na}$
<activity_defn> → c <activity_id> <coeff_of_variation>
/* Sqr. Coeff. of Var. */
| f <activity_id> <ph_type_flag> /* Phase type */
| h <entry_id> <hist_min> ':' <hist_max>
<hist_bins> <hist_type>

| M <activity_id> <max_service_time>
| s <activity_id> <ph_serv_time> /* Service time */
| Z <activity_id> <think_time> /* Think time */
| i <activity_id> <to_entry> <fan_in>
| o <activity_id> <to_entry> <fan_out>
| y <activity_id> <to_entry> <rendezvous> /* Rendezvous */
| z <activity_id> <to_entry> <send_no_reply>
/* Send-no-reply */

<activity_connections> → : <activity_conn_list> /* Activity Connections */
<activity_conn_list> → <activity_conn> { ; <activity_conn> }$_1^{na}$
<activity_conn> → <join_list>
| <join_list> -> <fork_list>
<join_list> → <reply_activity>
| <and_join_list>
| <or_join_list>
<fork_list> → <activity_id>
| <and_fork_list>
| <or_fork_list>
| <loop_list>
<and_join_list> → <reply_activity> { & <reply_activity>}$_1^{na}$
<quorum_count>$_{opt}$
<or_join_list> → <reply_activity> { + <reply_activity>}$_1^{na}$
<and_fork_list> → <activity_id> { & <activity_id>}$_1^{na}$
<or_fork_list> → <prob_activity> { + <prob_activity>}$_1^{na}$
<loop_list> → <loop_activity> { , <loop_activity>}$_0^{na}$ <end_activity>$_{opt}$
<prob_activity> → (<real>) <activity_id>
<loop_activity> → <real> * <activity_id>
<end_activity> → , <activity_id>
<reply_activity> → <activity_id> <reply_list>$_{opt}$
<reply_list> → [<entry_id> { , <entry_id> }$_0^{ne}$]
<quorum_count> → (<integer>) /* Quorum */

8. Expressions

<integer> → <int>
| { <expression> } /* integer result only */
<real> → <double>
| { <expression> }
<expression> → <expression> + <term>
| <expression> - <term>
| <term>
<term> → <term> * <factor>
| <term> / <factor>
| <term> % <factor> /* Modulus */
| <factor>
<factor> → <primary> ^ <factor>

```
                                                    /* Exponentiation, right associative */
                  |   <primary>
<primary>         →   + <atom>
                  |   - <atom>
                  |   <atom>
<atom>            →   ( <expression> )
                  |   <double>
<int>             →                                 /* Non negative integer */
<double>          →                                 /* Non negative double precision number */
```

9. Identifiers

Identifiers may be zero or more leading underscores '_', followed by a character, followed by any number of characters, numbers or underscores. Punctuation characters and other special characters are not allowed.