



Jaypee University of Information Technology
Solan (H.P.)
LEARNING RESOURCE CENTER

Acc. Num. SP02068 Call Num:

General Guidelines:

- ◆ Library books should be used with great care.
- ◆ Tearing, folding, cutting of library books or making any marks on them is not permitted and shall lead to disciplinary action.
- ◆ Any defect noticed at the time of borrowing books must be brought to the library staff immediately. Otherwise the borrower may be required to replace the book by a new copy.
- ◆ The loss of LRC book(s) must be immediately brought to the notice of the Librarian in writing.

Learning Resource Centre-JUIT



SP02068

OPERATING SYSTEM SCHEDULER FOR A DUAL CORE PROCESSOR

By

Manu Bhardwaj 021014

Niramay 021038

Mr. Vivek Sehgal,

Lecturer, Dept. of ECE



विद्या तत्र ज्योतिषमः

JAYPEE UNIVERSITY OF
INFORMATION TECHNOLOGY



MAY 2006

**Submitted in partial fulfillment of the requirements of the degree of
Bachelor of Technology**

DEPARTMENT OF ELECTRONICS

AND COMMUNICATION ENGINEERING

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY

WAKNAGHAT

CERTIFICATE

This is to certify that the work entitled, "Operating System Scheduler for a Dual Core Processor" submitted by Manu Bhardwaj (021014) and Niramay (021038) in partial fulfillment for the award of degree of Bachelor of Technology in Department of Electronics and Communication Engineering of Jaypee University of Information Technology has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Vivek Sehgal =

Mr. Vivek Sehgal,
Lecturer, Dept. of ECE

ACKNOWLEDGMENT

The authors wish to express their sincere appreciation for Mr. Vivek Sehgal, Lecturer, Dept. of Electronics & Communication Engineering for his guidance and whose familiarity with the needs and ideas of this topic was helpful during the programming phase of this project. We are also grateful to Prof. S. V. Bhooshan, H.o.D, Dept. of ECE, for giving us an opportunity to undertake this project. We are also indebted to this temple of learning, from where we have completed our course, for providing us with a well stocked library and a modern computer laboratory where we both have spent countless hours poring over books and screens trying to find out that elusive bug or deciding our next step in the algorithm.

TABLE OF CONTENTS

i. Certificate.....	3
ii. Acknowledgement.....	4
iii. Abstract.....	10
1. Chapter One - Project Description.....	11
1.1. Motivation.....	11
1.2. Statement of Problem.....	11
2. Chapter Two - O.S. Scheduler.....	13
2.1. Scheduling.....	13
2.2. Types of operating system schedulers.....	14
2.3. Optimization Criteria for A Scheduler.....	16
2.4. Process Control Block.....	16
2.5. Dispatcher.....	17
2.6. Interrupts.....	17
2.7. Types of Scheduling.....	18
2.8. Round Robin Scheduling Algorithm.....	30
2.9. Types of Processes Handled by CPU.....	32
2.10. Performance of Round Robin Scheduler for Single Execution Core.....	32
2.11. Properties of Round Robin.....	33
2.12. Rule of thumb In Case of Round Robin Implementation.....	33
3. Chapter Three – Dual Core Processors.....	34
3.1. Introduction.....	34
3.2. Multi Core System.....	35
3.3. Commercial Example.....	35
3.4. Development Motivation.....	36
3.5. Advantages of Dual Core Processor.....	37
3.6. Disadvantages.....	38
3.7. Software Development Considerations.....	38
3.8. Software Impact.....	39

3.9.Licensing.....	39
4. Chapter Four – Single Core Scheduler Software Development.....	41
4.1. Steps in Development.....	41
4.2. Requirement Specification.....	42
4.3. High Level Design.....	42
4.4. Detailed Design.....	43
4.5. Coding.....	46
4.6. Software Testing.....	49
4.7. Software Pitfalls to Avoid.....	54
4.8. Algorithm/Implementation.....	57
4.9. Research Instruments/Tools.....	58
4.10. Snapshots of the Software Interface.....	59
5. Chapter Five – Dual Core Scheduler Software Development.....	61
5.1. Steps in Development.....	61
5.2. Requirement Specification.....	62
5.3. High Level Design.....	62
5.4. Detailed Design.....	63
5.5. Coding.....	66
5.6. Software Testing.....	70
5.7. Software Pitfalls to Avoid.....	75
5.8. Algorithm/Implementation.....	78
5.9. Research Instruments/Tools.....	79
5.10. Snapshots of the Software Interface.....	80
6. Chapter Four – Graphical Utility Software Development.....	82
6.1. Steps in Development.....	82
6.2. Requirement Specification.....	83
6.3. High Level Design.....	83
6.4. Detailed Design.....	84
6.5. Coding.....	85
6.6. Software Testing.....	86
6.7. Software Pitfalls to Avoid.....	89

6.8. Algorithm/Implementation.....	92
6.9. Research Instruments/Tools.....	92
7. Conclusion.....	94
8. Bibliography.....	95
9. Glossary.....	96

LIST OF FIGURES

1. Process Control Block.....	16
2. Dispatcher.....	17
3. Interrupt.....	17
4. Block Diagram of Dual Core.....	34
5. Snapshot of Main Application Window.....	59
6. Snapshot of output generated by the software.....	60
7. Snapshot of Main Application Window.....	80
8. Snapshot of output generated by the software.....	81

LIST OF ABBREVIATIONS

1. AMD..... Advanced Micro Devices
2. CPU..... Central Processing Unit
3. PC..... Personal Computer
4. OS..... Operating System
5. I/O..... Input/Output
6. FCFS..... First-Come, First-Served
7. SJF..... Shortest Job First
8. DRR..... Deficit Round Robin
9. EDF..... Earliest Deadline First
10. FIFO..... First In, First Out

ABSTRACT

Scheduling is a key concept in operating system design. It refers to the way processes are assigned priorities in a priority queue. This assignment is carried out by software known as a scheduler. In general-purpose operating systems, the goal of the scheduler is to balance processor loads, and prevent any one process from either monopolizing the processor or being starved for resources. In this project we have developed a new operating system scheduler for a dual execution core general purpose processor and then compared its performance with the more common single execution core processor scheduler based on the round robin algorithm. This comparison is done by a graphical utility developed by the authors which extracts the parameters of the schedulers and generates graphs for various categories. We show, in particular, that the scheduler for the dual core processor gives a better performance than the one for the single execution core.

CHAPTER ONE

PROJECT DESCRIPTION

1.1 Motivation

The idea of developing our own operating system scheduler bore fruit when the authors were studying the course Embedded Systems taught by the project guide himself in the 7th semester of our B.Tech. course. The course contained a significant part of operating system theory. During that same time both Intel and AMD launched their respective dual core CPU's in the market. This caught the attention of the authors and acted as a catalyst to the authors thinking. We combined both and thus came up with the proposal of developing an operating system scheduler for a dual core processor.

1.2 Statement of problem

The coupling of execution cores in a processor results in simultaneous management of activities. In a traditional processor, in case of multi-tasking, the processor must switch back and forth between two or more sets of data streams and programs. This way the CPU resources are depleted and its performance suffers. In a dual core processor each core handles incoming data strings simultaneously to improve efficiency. Now when one is executing the other can be accessing the system bus or executing its own code. Thus having multiple execution cores dramatically increases the PC's capabilities and computing resources, which reflects a shift to better responsiveness, higher multithreaded throughput and ultimately, parallel computing. In developing a scheduler for the dual core processor the developer must worry about how to efficiently use the multiple cores while still preserving application level quality of service. Ideally, the multiple execution cores must have equal load sharing. We also investigate the performance of the operating system scheduler for a dual core processor using quality of service parameters (such as average waiting time for a task in ready queue, load comparison between the two cores, percentage of busy time of the CPU, etc.). The parameters are then compared with that of an operating system scheduler for a single core processor and it is verified that the dual core scheduler gives better performance in most conditions. The comparison is done by

developing another software utility with a graphical interface which can generate a graph on screen by retrieving the parameters of both the schedulers.

CHAPTER TWO

OS SCHEDULER

2.1 Scheduling

Scheduling is a key concept in computer multitasking and multiprocessing operating system design, and in real-time operating system design. It refers to the way processes are assigned priorities in a priority queue. This assignment is carried out by software known as a scheduler. On most multitasking system, only one process can truly be active at a time-the system must therefore share its time between the executions of many processes. This sharing is called Scheduling. The CPU scheduler selects a process from among the ready processes to execute on the CPU. CPU scheduling is the basis for multiprogrammed operating systems. CPU utilization increases by switching the CPU among ready processes instead of waiting for each process to terminate before executing the next. The idea of multiprogramming could be described as follows: A process is executed by the CPU until it completes or goes for an I/O. In simple systems with no multiprogramming the CPU is idle till the process completes the I/O and restarts execution. With multiprogramming, many ready processes are maintained in memory. So when CPU becomes idle as in the case above, the operating system switches to execute another process each time a current process goes into a wait for I/O. A CPU scheduler selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

In general-purpose operating systems, the goal of the scheduler is to balance processor loads, and prevent any one process from either monopolizing the processor or being starved for resources. The built-in schedulers in Microsoft Windows have no provision for preventing the monopolizing of a processor, or preventing resource starvation by a process. If a process (by design or accident) should attempt to aggressively grab all available processor resources, the system often reacts by slowing down to the point of being perceived as "locked". In operating systems designed from the

ground up for critical application use, such as Solaris, AIX or zOS, schedulers include these "defensive" features. There are third-party applications for Windows that add these features, for example AppSense Performance Manager, Citrix Presentation Server 4.0 and Aurema ARMTech.

In real-time environments, such as devices for automatic control in industry (for example robotics), the scheduler also must ensure that processes can meet deadlines; this is crucial for keeping the system stable.

2.2 Types of operating system schedulers

Operating Systems may feature up to 3 distinct types of schedulers:

- a long-term scheduler (also known as an admission scheduler),
- a mid-term or medium-term scheduler and
- a short-term scheduler (also known as a dispatcher).

The long-term, or admission, scheduler decides which jobs or processes are to be admitted to the "ready" queue; that is, when an attempt is made to execute a program, its admission to the set of currently executing processes is either authorized or delayed by the long-term scheduler. Thus, this scheduler dictates what processes are to run on a system and the degree of concurrency to be supported at any one time - ie: whether a high or low amount of processes are to be executed concurrently, and how the split between IO intensive and CPU intensive processes is to be handled. Typically for a desktop computer, there is no long-term scheduler as such, and processes are admitted to the system automatically. However this type of scheduling is very important for a real time system, as the systems ability to meet process deadlines may be compromised by the slowdowns and contention resulting from the admission of more processes than the system can safely handle.

The mid-term scheduler, present in all systems with virtual memory, temporarily removes processes from main memory and places them on secondary memory (such as a disk drive) or vice versa. This is commonly referred to as "swapping out" or "swapping in" (also incorrectly as "paging out" or "paging in"). The mid-term scheduler may decide to swap out a process which has not been active for some time, or a process which has a low priority, or a process which is page faulting frequently, or a process which is taking up a large amount of memory in order to free up main memory for other processes, swapping the process back in later when more memory is available, or when the process has been unblocked and is no longer waiting for a resource.

In many systems today (those that support mapping virtual address space to secondary storage other than the swap file), the mid-term scheduler may actually perform the role of the long-term scheduler, by treating binaries as "swapped out processes" upon their execution. In this way, when a segment of the binary is required it can be swapped in on demand, or "lazy loaded".

The short-term scheduler (also known as the dispatcher) decides which of the ready, in memory processes are to be executed (allocated a CPU) next, following a clock interrupt, an IO interrupt, an operating system call or another form of signal. Thus the short-term scheduler makes scheduling decisions much more frequently than the long-term or mid-term schedulers - a scheduling decision will at a minimum have to be made after every time slice and these are very short. This scheduler can be preemptive, implying that it is capable of forcibly removing processes from a CPU when it decides to allocate that CPU to another process, or non-preemptive, in which case the scheduler is unable to "force" processes off the CPU.

CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state.
2. Switches from running to ready state.
3. Switches from waiting to ready.

4. Terminates.

Scheduling under 1 and 4 is non preemptive. All other scheduling is preemptive.

2.3 Optimization Criteria for A Scheduler

- Max CPU utilization : Keep the CPU as busy as possible.
- Max throughput : Maximum processing done at a time.
- Min turnaround time : Minimum amount of time to execute a particular process.
- Min waiting time :A process should wait minimum amount of time in the ready queue.
- Min response time :It should take minimum amount of time from when a request was submitted until the first response is produced.

2.4 Process Control Block

A Process Control Block(PCB, also called Task Control Block) data structure in the operating system kernel representing the state of a given process.

It includes:

- Process id
- Process State (state diagram)
- Registers (and program counter).
- Memory info.
- List of open files, Inter process communication info.
- Accounting info.
- Pointers to other data structures in the OS.

Process Control Block

◆ Contains state information such as:

• Process State	Ready, etc
• Process ID	to relate I/O, events and process communications (P.X) etc.
• Priority	or time slot, time to run etc.
• Memory pointers	location of process code and data MMU registers
• Resources Allocated	terminals, devices
• Register Save Area	processor registers, stack pointer etc.
• Owner	User ID
• Parent	Parent process ID

◆ Scheduling involves saving volatile registers in one PCB and restoring them from another

Figure No.1 Process Control Block

2.5 Dispatcher

Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:

This includes:

- switching context
- switching to user mode
- jumping to the proper location in the user program to restart that program

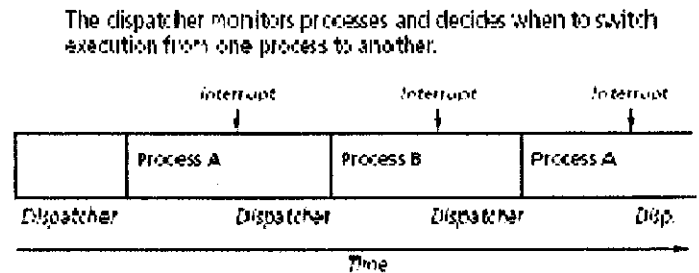


Figure No.2 Dispatcher

2.6 Interrupts

Interrupts are asynchronous breaks in program flow that occur as a result of events outside the running program. They are usually hardware related, stemming from events such as a button press, timer expiration, or completion of a data transfer. We can see from these examples that interrupt conditions are independent of particular instructions; they can happen at any time. Interrupts trigger execution of instructions that perform work on behalf of the system, but not necessarily the current program.

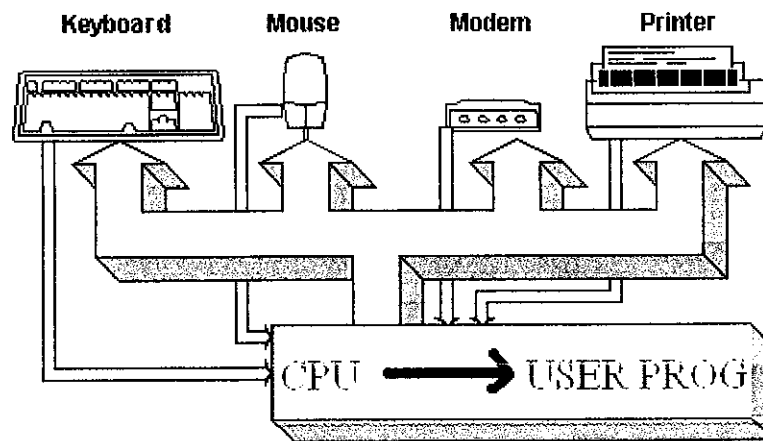


Figure No.3 Interrupt

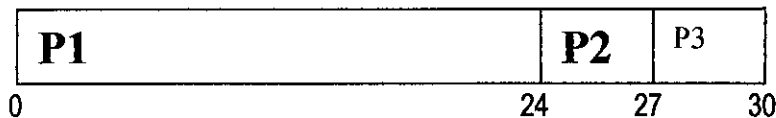
2.7 Types of Scheduling

2.7.1 First-Come, First-Served (FCFS) Scheduling

This is one of the very brute force algorithms. A process that requests for the CPU first is allocated the CPU first. Hence the name first come first serve. The FCFS algorithm is implemented by using a first-in-first-out (FIFO) queue structure for the ready queue. This queue has a head and a tail. When a process joins the ready queue its PCB is linked to the tail of the FIFO queue. When the CPU is idle, the process at the head of the FIFO queue is allocated the CPU and deleted from the queue. Even though the algorithm is simple, the average waiting is often quite long and varies substantially if the CPU burst times vary greatly as seen in the following example. Consider a set of three processes P1, P2 and P3 arriving at time instant 0 and having CPU burst times as shown below:

Process	Burst time (msecs)
P1	24
P2	3
P3	3

The Gantt chart below shows the result.



Average waiting time and average turnaround time are calculated as follows:

The waiting time for process P1 = 0 msecs

P2 = 24 msecs

P3 = 27 msecs

Average waiting time = $(0 + 24 + 27) / 3 = 51 / 3 = 17$ msecs.

P1 completes at the end of 24 msec, P2 at the end of 27 msec and P3 at the end of 30 msec. Average turnaround time = $(24 + 27 + 30) / 3 = 81 / 3 = 27$ msec.

If the processes arrive in the order P2, P3 and P1, then the result will be as follows:



Average waiting time = $(0 + 3 + 6) / 3 = 9 / 3 = 3$ msec.

Average turnaround time = $(3 + 6 + 30) / 3 = 39 / 3 = 13$ msec.

Thus if processes with smaller CPU burst times arrive earlier, then average waiting and average turnaround times are lesser.

The algorithm also suffers from what is known as a convoy effect.

Consider the following scenario:

1. Let there be a mix of one CPU bound process and many I/O bound processes in the ready queue.
2. The CPU bound process gets the CPU and executes (long I/O burst).
3. In the meanwhile, I/O bound processes finish I/O and wait for CPU thus leaving the I/O devices idle.
4. The CPU bound process releases the CPU as it goes for an I/O.
5. I/O bound processes have short CPU bursts and they execute and go for I/O quickly. The CPU is idle till the CPU bound process finishes the I/O and gets hold of the CPU.
6. The above cycle repeats. This is called the convoy effect. Here small processes wait for one big process to release the CPU.
7. Since the algorithm is non preemptive in nature, it is not suited for time sharing systems.

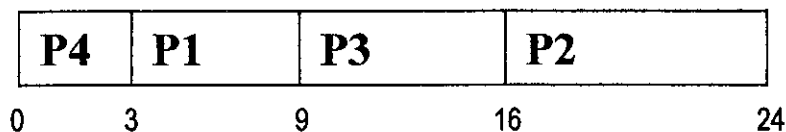
2.7.2 Shortest-Job-First

Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time. Another approach to CPU scheduling

is the shortest job first algorithm. In this algorithm, the length of the CPU burst is considered. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. Hence the name- shortest job first. In case there is a tie, FCFS scheduling is used to break the tie. As an example, consider the following set of processes P1, P2, P3, P4 and their CPU burst times:

Process	Burst time (msecs)
P1	6
P2	8
P3	7
P4	3

Using SJF algorithm, the processes would be scheduled as shown below.



Average waiting time = $(0 + 3 + 9 + 16) / 4 = 28 / 4 = 7$ msecs.

Average turnaround time = $(3 + 9 + 16 + 24) / 4 = 52 / 4 = 13$ msecs.

If the above processes were scheduled using FCFS algorithm, then

Average waiting time = $(0 + 6 + 14 + 21) / 4 = 41 / 4 = 10.25$ msecs.

Average turnaround time = $(6 + 14 + 21 + 24) / 4 = 65 / 4 = 16.25$ msecs.

The SJF algorithm produces the most optimal scheduling scheme. For a given set of processes, the algorithm gives the minimum average waiting and turnaround times. This is because, shorter processes are scheduled earlier than longer ones and hence waiting time for shorter processes decreases more than it increases the waiting time of long processes.

The main disadvantage with the SJF algorithm lies in knowing the length of the next CPU burst. In case of long-term or job scheduling in a batch system, the time

required to complete a job as given by the user can be used to schedule. SJF algorithm is therefore applicable in long-term scheduling.

The algorithm cannot be implemented for CPU scheduling as there is no way to accurately know in advance the length of the next CPU burst. Only an approximation of the length can be used to implement the algorithm.

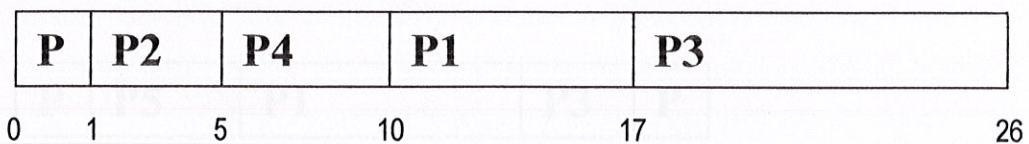
But the SJF scheduling algorithm is provably optimal and thus serves as a benchmark to compare other CPU scheduling algorithms.

SJF algorithm could be either preemptive or non preemptive. If a new process joins the ready queue with a shorter next CPU burst than what is remaining of the current executing process, then the CPU is allocated to the new process. In case of non preemptive scheduling, the current executing process is not preempted and the new process gets the next chance, it being the process with the shortest next CPU burst.

Given below are the arrival and burst times of four processes P1, P2, P3 and P4.

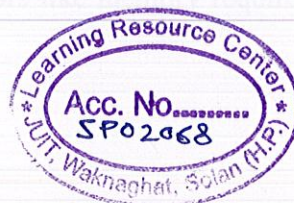
Process	Arrival time (msecs)	Burst time (msecs)
P1	0	8
P2	1	4
P3	2	9
P4	3	5

If SJF preemptive scheduling is used, the following Gantt chart shows the result.



Average waiting time = $((10 - 1) + 0 + (17 - 2) + (15 - 3)) / 4 = 26 / 4 = 6.5$ msec.

If non preemptive SJF scheduling is used, the result is as follows:



P1	P2	P4	P3
0	8	12	17
26			

Average waiting time = $((0 + (8 - 1) + (12 - 3) + (17 - 2)) / 4 = 31 / 4 = 7.75$ msec.

2.7.3 Priority Scheduling

Each process can be associated with a priority. CPU is allocated to the process having the highest priority. Hence the name- priority. Equal priority processes are scheduled according to FCFS algorithm.

The SJF algorithm is a particular case of the general priority algorithm. In this case priority is the inverse of the next CPU burst time. Larger the next CPU burst, lower is the priority and vice versa. In the following example, we will assume lower numbers to represent higher priority.

Process	Priority	Burst time (msecs)
P1	3	10
P2	1	1
P3	3	2
P4	4	1
P5	2	5

Using priority scheduling, the processes are scheduled as shown below:

P	P5	P1	P3	P
0	1	6	16	18
19				

Average waiting time = $(6 + 0 + 16 + 18 + 1) / 5 = 41 / 5 = 8.2$ msec.

Priorities can be defined either internally or externally. Internal definition of priority is based on some measurable factors like memory requirements, number

of open files, and so on. External priorities are defined by criteria such as importance of the user depending on the user's department and other influencing factors.

Priority based algorithms can be either preemptive or nonpreemptive. In case of preemptive scheduling, if a new process joins the ready queue with a priority higher than the process that is executing, then the current process is preempted and CPU allocated to the new process. But in case of nonpreemptive algorithm, the new process having highest priority from among the ready processes is allocated the CPU only after the current process gives up the CPU.

Starvation or indefinite blocking is one of the major disadvantages of priority scheduling. Every process is associated with a priority. In a heavily loaded system, low priority processes in the ready queue are starved or never get a chance to execute. This is because there is always a higher priority process ahead of them in the ready queue.

A solution to starvation is aging. Aging is a concept where the priority of a process waiting in the ready queue is increased gradually. Eventually even the lowest priority process ages to attain the highest priority; at which time, it gets a chance to execute on the CPU.

2.7.4 Deficit Round Robin (DRR) (also deficit weighted round robin)

It is a modified weighted round robin scheduling discipline. DRR was proposed by M. Shreedhar and G. Varghese in 1995. It can handle packets of variable size without knowing their mean size. A maximum packet size number is subtracted from the packet length, and packets that exceed that number are held back until the next visit of the scheduler. WRR serves every nonempty queue whereas DRR serves packets at the head of every nonempty queue which deficit counter is greater than the packet's size. If it is lower then deficit counter is increased by some given value called quantum. Deficit counter is decreased by the size of packets being served. The complexity of DRR is $O(1)$.

2.7.5 Earliest Deadline First Scheduling

Earliest deadline first (EDF) scheduling is a dynamic scheduling principle used in real-time operating systems. It places processes in a priority queue. Whenever a scheduling event occurs (task finishes, new task released, etc.) the queue will be searched for the process closest to its deadline. This process will then be scheduled for execution next. With scheduling periodic processes that have deadlines equal to their periods, EDF has a utilization bound of 100%. That is, EDF can guarantee that all deadlines are met provided that the total CPU utilization is not more than 100%. So, compared to fixed priority scheduling techniques like rate-monotonic scheduling, EDF can guarantee all the deadlines in the system at higher loading. However, when the system is overloaded, the set of processes that will miss deadlines is largely unpredictable (it will be a function of the exact deadlines and time at which the overload occurs.) This is a considerable disadvantage to a real time systems designer. The algorithm is also difficult to implement in hardware and there is a tricky issue of representing deadlines in different ranges (deadlines must be rounded to finite amounts, typically a few bytes at most). Therefore EDF is not commonly found in industrial real-time computer systems. There is a significant body of research dealing with EDF scheduling in real-time computing; it is possible to calculate worst case response times of processes in EDF, to deal with other types of processes than periodic processes and to use servers to regulate overloads.

2.7.6 Fair-Share Scheduling

Fair-share scheduling is a scheduling strategy for computer operating systems in which the CPU usage is equally distributed among system users or groups, as opposed to equal distribution among processes. For example, if four users (A,B,C,D) are concurrently executing one process each, the scheduler will logically divide the available CPU cycles such that each user gets 25% of the whole ($100\% / 4 = 25\%$). If user B starts a second process, each user will still receive 25% of the total cycles, but both of user B's processes will now use 12.5%. On the other hand, if a new user starts a process on the system, the

scheduler will reapportion the available CPU cycles such that each user gets 20% of the whole ($100\% / 5 = 20\%$).

Another layer of abstraction allows us to partition users into groups, and apply the fair share algorithm to the groups as well. In this case, the available CPU cycles are divided first among the groups, then among the users within the groups, and then among the processes for that user. For example, if there are three groups (1,2,3) containing three, two, and four users respectively, the available CPU cycles will be distributed as follows:

- * $100\% / 3 \text{ groups} = 33.3\% \text{ per group}$
- * Group 1: $(33.3\% / 3 \text{ users}) = 11.1\% \text{ per user}$
- * Group 2: $(33.3\% / 2 \text{ users}) = 16.7\% \text{ per user}$
- * Group 3: $(33.3\% / 4 \text{ users}) = 8.3\% \text{ per user}$

One common method of logically implementing the fair-share scheduling strategy is to recursively apply the round-robin scheduling strategy at each level of abstraction (processes, users, groups, etc.) The time quantum required by round-robin is arbitrary, as any equal division of time will produce the same results. Moab Cluster Suite is a cluster, grid and HPC scheduler that uses Fair-share capabilities to distribute compute resources.

2.7.7 Gang Scheduling

In Computer science, Gang scheduling is a scheduling algorithm that schedules related threads or processes to run simultaneously on different processors. Usually these will be threads all belonging to the same process, but they may also be from different processes, for example when the processes have a producer-consumer relationship, or when they all come from the same MPI program.

Gang scheduling is used so that if two threads or processes communicate with each other, they will all be ready to communicate at the same time. If they were

not gang-scheduled, then one could wait to send or receive a message to another while it is sleeping, and vice-versa. When processors are over-subscribed and gang scheduling is not used within a group of processes or threads which communicate with each other, it can lead to starvation.

2.7.8 Least Slack Time Scheduling

Least Slack Time (LST) scheduling is a scheduling algorithm. It assigns priority based on the slack time of a process. It is also known as Least Laxity First. Its most common use is in embedded systems, especially those with multiple processors. It imposes the simple constraint that each process on each available processor possesses the same run time, and that individual processes do not have an affinity to a certain processor. This is what lends it suitability to embedded systems. This scheduling algorithm first selects those processes that have the smallest "slack time". Slack time is defined as the temporal difference between the deadline, the ready time and the run time.

More formally, the slack time for a process is defined as:

$$(d - t) - c'$$

Where d is the process deadline, t is the real time since the cycle start, and c' is the remaining computation time. Thus, this algorithm tries to schedule each process as late as possible.

LST scheduling is most useful in systems comprising mainly aperiodic tasks, because no prior assumptions are made on the events' rate of occurrence. The main weakness of LST is that it does not look ahead, and works only on the current system state. Thus, during a brief overload of system resources, LST can be sub-optimal. It will also be suboptimal when used with uninterruptible processes. However, like earliest deadline first, and unlike rate monotonic scheduling, this algorithm can be used for processor utilization up to 100%.

2.7.9 List Scheduling

The basic idea of list scheduling is to make an ordered list of processes by assigning them some priorities, and then repeatedly execute the following two steps until a valid schedule is obtained:

- * Select from the list, the process with the highest priority for scheduling.
- * Select a resource to accommodate this process.

The priorities are determined statically before scheduling process begins. The first step chooses the process with highest priority, the second step select the best possible resource. Some known list scheduling strategies are :

- * Highest Level First algorithm or HLF
- * Longest Path algorithm or LP
- * Longest Processing Time
- * Critical Path

2.7.10 Lottery Scheduling

Lottery Scheduling is a probabilistic scheduling algorithm for processes in an operating system. Processes are each assigned some number of lottery tickets, and the scheduler draws a random ticket to select the next process. The distribution of tickets need not be uniform; granting a process more tickets provides it a relative higher chance of selection. This technique can be used to approximate other scheduling algorithms, such as shortest job next and Fair-share scheduling.

Lottery scheduling solves the problem of starvation. Giving each process at least one lottery ticket guarantees that it has non-zero probability of being selected at each scheduling operation.

2.7.11 Multilevel Feedback Queue

In computer science, a multilevel feedback queue is a scheduling algorithm designed by Leonard Kleinrock in 1970. It is intended to meet the following design requirements for multimode systems:

1. Give preference to short jobs.
2. Give preference to I/O bound processes.
3. Quickly establish the nature of a process and schedule the process accordingly.

Multiple FIFO queues are used and the operation is as follows;

1. A new process is positioned at the end of the top-level FIFO queue.
2. At some stage the process reaches the head of the queue and is assigned the CPU.
3. If the process is completed it leaves the system.
4. If the process voluntarily relinquishes control it leaves the queuing network, and when the process becomes ready again it enters the system on the same queue level.
5. If the process uses all the quantum time, it is pre-empted and positioned at the end of the next lower level queue.
6. This will continue until the process completes or it reaches the base level queue.

At the base level queue the processes circulate in round robin fashion until they complete and leave the system. In the multilevel feedback queue a process is given just one chance to complete at a given queue level before it is forced down to a lower level queue.

2.7.12 Proportional Share Scheduling

Proportional Share Scheduling is a type of scheduling which preallocates certain amount of CPU time to each of the processes.

2.7.13 Two-Level Scheduling

Two-level scheduling is a computer science term to describe a method to more efficiently perform process scheduling that involves swapped out processes.

Consider this problem: A system contains 50 running processes all with equal priority. However, the system's memory can only hold 10 processes in memory simultaneously. Therefore, there will always be 40 processes swapped out written on virtual memory on the hard disk. The time taken to swap out and swap in a process is 50 ms respectively.

With straightforward Round-robin scheduling, every time a context switch occurs, there would be an 80% probability (40/50, if it chooses randomly among the processes) that a process would need to be swapped in. If that occurs, then obviously a process also needs to be swapped out. Swapping in and out of memory is costly, and the scheduler would waste much of its time doing unneeded swaps.

That is where two-level scheduling enters the picture. It uses two different schedulers, one lower-level scheduler which can only select among those processes in memory to run. That scheduler could be a Round-robin scheduler. The other scheduler is the higher-level scheduler whose only concern is to swap in and swap out processes from memory. It does its scheduling much less often than the lower-level scheduler since swapping takes so much time.

Thus, the higher-level scheduler selects among those processes in memory that have run for a long time and swaps them out. They are replaced with processes on disk that have not run for a long time. Exactly how it selects processes is up to the implementation of the higher-level scheduler. A compromise has to be made involving the following variables:

- * Response time: A process should not be swapped out for too long. Then some other process (or the user) will have to wait needlessly long. If this variable is not considered resource starvation may occur and a process may not complete at all.
- * Size of the process: Larger processes must be subject to fewer swaps than smaller ones because they take longer time to swap. Because they are larger, fewer processes can share the memory with the process.
- * Priority: The higher the priority of the process, the longer it should stay in memory so that it completes faster.

2.8 Round Robin Scheduling Algorithm

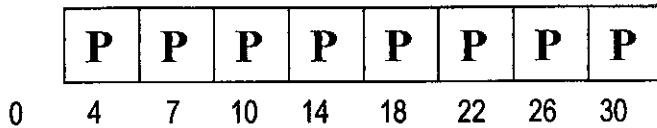
The round-robin CPU scheduling algorithm is basically a preemptive scheduling algorithm designed for time-sharing systems. One unit of time is called a time slice. Duration of a time slice may range between 10 msec. and about 100 msec. The CPU scheduler allocates to each process in the ready queue one time slice at a time in a round-robin fashion. Hence the name- round-robin.

The ready queue in this case is a FIFO queue with new processes joining the tail of the queue. The CPU scheduler picks processes from the head of the queue for allocating the CPU. The first process at the head of the queue gets to execute on the CPU at the start of the current time slice and is deleted from the ready queue. The process allocated the CPU may have the current CPU burst either equal to the time slice or smaller than the time slice or greater than the time slice. In the first two cases, the current process will release the CPU on its own and there by the next process in the ready queue will be allocated the CPU for the next time slice. In the third case, the current process is preempted, stops executing, goes back and joins the ready queue at the tail there by making way for the next process.

Consider the same example explained under FCFS algorithm.

Process	Burst time (msecs)
P1	24
P2	3

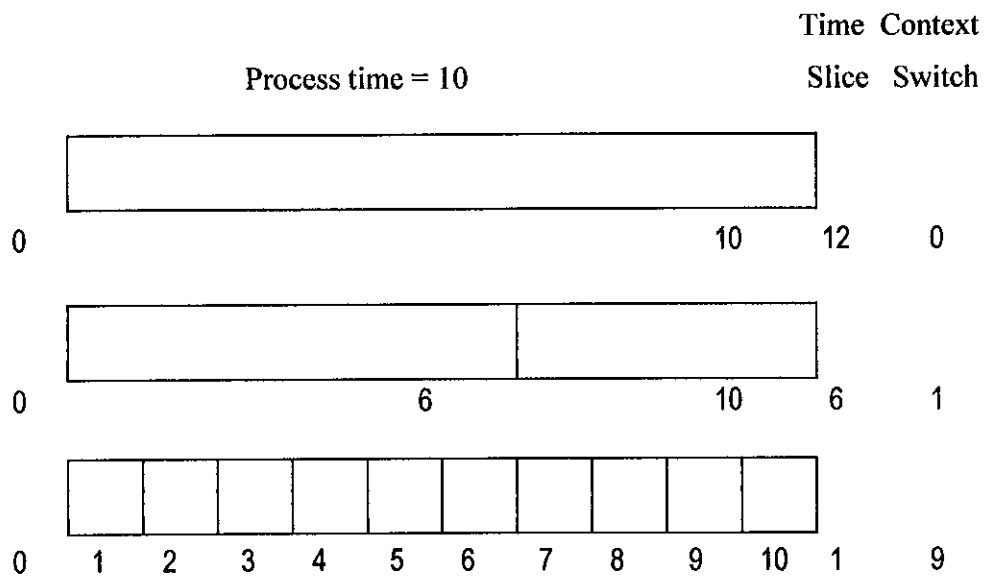
Let the duration of a time slice be 4 msec, which is to say CPU switches between processes every 4 msec in a round-robin fashion. The Gantt chart below shows the scheduling of processes.



Average waiting time = $(4 + 7 + (10 - 4)) / 3 = 17 / 3 = 5.66$ msec.

If there are 5 processes in the ready queue that is $n = 5$, and one time slice is defined to be 20 msec that is $q = 20$, then each process will get 20 msec or one time slice every 100 msec. Each process will never wait for more than $(n - 1) \times q$ time units.

The performance of the RR algorithm is very much dependent on the length of the time slice. If the duration of the time slice is indefinitely large then the RR algorithm is the same as FCFS algorithm. If the time slice is too small, then the performance of the algorithm deteriorates because of the effect of frequent context switching. Below is shown a comparison of time slices of varying duration and the context switches they generate on only one process of 10 time units.



The above example shows that the time slice should be large with respect to the context switch time else if RR scheduling is used the CPU will spend more time in context switching.

2.9 Types of Processes Handled by CPU

- IO Bound processes: processes that perform lots of IO operations. Each IO operation is followed by a short CPU burst to process the IO, and then more IO happens.
- CPU bound processes: processes that perform lots of computation and do little IO. Tend to have a few long CPU bursts.

2.10 Performance of Round Robin Scheduler for Single Execution Core

Let q be the time quantum, then

- q Large \Rightarrow FIFO
- q Small $\Rightarrow q$ must be large with respect to context switch, otherwise overhead is too high.

The most interesting issue with round robin scheme is the *length of the quantum*. Setting the quantum too short causes too many context switches and lower the CPU efficiency. On the other hand, setting the quantum too long may cause poor response time and approximates First Come First Served (FCFS).

Let's assume that task switching takes 2 msecs.

If we have a quantum of 8 msecs, we ensure very good response time. For example, imagine 20 users all logged in to a single CPU server; with every user making a request at the same time, each task takes up 10 msecs (8 msecs quantum + 2 msecs overhead),

and the 20th user gets a response in **200** msec (10 msec*20), which is pretty good (5th second).

On the other hand, efficiency is:

$$\text{Useful time} \times \text{total time} = 8\text{ms} \times 10\text{ms} = 80\%$$

i.e. 20% of the CPU time is wasted on overhead.

With a 200 msec quantum, efficiency is 200 msec / 202 msec = ~99%

But, response time if 20 users make a request at once is $202 * 20 = 4040$ msec or > 4 seconds, which is not good.

2.11 Properties of Round Robin

1. Advantages: simple, low overhead, works for interactive systems
2. Disadvantages: if quantum is too small, too much time wasted in context switching; if too large (i.e. longer than mean CPU burst), approaches FCFS.
3. Typical value: 20 – 40 msec

2.12 Rule of thumb In Case of Round Robin Implementation

Choose *quantum* so that large majority (80 – 90%) of jobs finish CPU burst in one quantum.

CHAPTER THREE

DUAL CORE PROCESSORS

3.1 Introduction

Dual-core and multi-core processors are designed by including two or more full execution cores within a single processor, enabling simultaneous management of activities. Imagine that a dual-core processor is like a four-lane highway—it can handle up to twice as many cars as its two-lane predecessor without making each car drive twice as fast. Two identical processors are manufactured so they reside side-by-side on the same die. Each of the physical processor cores has its own resources (architectural state, registers, execution units, etc.). The multiple cores on-die may or may not share several layers of the on-die cache. Typically, this means that two identical processors are manufactured so they reside side-by-side on the same die. It is also possible to (vertically) stack two separate processor die and place them in the same IC package. Each of the physical processor cores has its own resources (architectural state, registers, execution units, etc.). The multiple cores on-die may or may not share several layers of the on-die cache. A dual core processor design could provide for each physical processor to: 1) have its own on-die cache, or 2) it could provide for the

on-die cache to be shared by the two processors, or 3) each processor could have a portion of on-die cache that is exclusive to a single processor and then have a portion of on-die cache that is shared between the two dual core processors. The two cores in a dual core package could have an on-die communication path between them so that putting snoops and requests out on the FSB is not necessary. Both processors must have a communication path to the computer system front-side bus. Note that dual core processors could also contain HT Technology which would enable a single processor IC package,

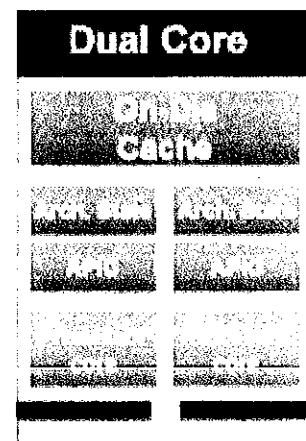


Figure No.1 Block Diagram of Dual Core

containing two physical processors, to appear as four logical processors capable of running four programs or threads simultaneously.

3.2 Multi Core System

The multi core system is an extension to the dual core system except that it would consist of more than 2 processors cores. The current trends in processor technology indicate that the number of processor cores in one IC chip will continue to increase. If we assume that the number of transistors per processor core remains relatively fixed, it is reasonable to assume that the number of processor cores could follow Moore's Law, which states that the number of transistors per a certain area on the chip will double approximately every 18 months. Even if this trend does not follow Moore's Law, the number of processor cores per chip appears destined to steadily increase - based on statements from several processor manufacturers. The optimal number of processors is yet to be determined, but will probably change over time as software adapts to effectively use many processors, simultaneously. However, a software program that is only capable of running on one processor (or very few processors) will be unable to take full advantage of future processors that contain many processors cores. For example, an application running on a 4-processor system with each socket containing quad-core processors has 16 processor cores available to schedule 16 program threads simultaneously.

3.3 Commercial examples

- International Business Machines (IBM)'s POWER4, first Dual-Core module processor released in 2000
- IBM's POWER5 dual-core chip is now in production, and the company has a PowerPC 970MP dual-core processor in production and is in use in the Apple PowerMac G5.

- PA-RISC (PA-8800)
- Sun Microsystems UltraSPARC IV, UltraSPARC IV+, UltraSPARC T1
- Intel's dual-core Xeon processors, code-named *Paxville* and *Dempsey*, are shipping at 3 GHz. The company is also currently developing dual-core versions of its Itanium high-end server CPU architecture but there have been many delays.
- AMD released its dual-core Opteron server/workstation processors on 22 April 2005, and its dual-core desktop processors, the Athlon 64 X2 family, were released on 31 May 2005. AMD have also recently released the FX-60.
- Motorola/Freescale has dual-core ICs based on the PowerPC e600 and e700 cores in development.
- Intel released the Core Duo processor in 2006. It is available in the Apple iMac, high end Mac mini and MacBook Pro, as well as in various laptop PCs, from brands of the likes of Sony, Toshiba, ASUS, and others.
- Microsoft's Xbox 360 game console.

3.4 Development Motivation

3.4.1 *Technical pressures*

While CMOS manufacturing technology continues to improve, reducing the size of single gates, physical limits of semiconductor-based microelectronics become a major design concern. Some effects of these physical limitations can cause significant heat dissipation and data synchronization problems. The demand for more complex and capable microprocessors causes CPU designers to utilize various methods of increasing performance. Some ILP methods like superscalar pipelining are suitable for many applications, but are inefficient for others that tend to contain difficult-to-predict code. Many applications are better suited to TLP methods, and multiple independent CPUs is one common method used to increase a system's overall TLP. A combination of increased available space due to refined manufacturing processes and the demand for increased TLP led to the logical creation of multi-core CPUs.

3.4.2 Commercial incentives

Several business motives drive the development of dual-core architectures. Since SMP designs have been long implemented using discrete CPUs, the issues regarding implementing the architecture and supporting it in software are well known. Additionally, utilizing a proven processing core design (e.g. Freescale's e700 core) without architectural changes reduces design risk significantly. Finally, the connotations of the terminology "dual-core" (and other multiples) lends itself to marketing efforts.

3.4.3 Frequency Bottlenecks

Additionally, for general-purpose processors, much of the motivation for multi-core processors comes from the increasing difficulty of improving processor performance by increasing the operating frequency (frequency-scaling). In order to continue delivering regular performance improvements for general-purpose processors, manufacturers such as Intel and AMD have turned to multi-core designs, sacrificing lower manufacturing costs for higher performance in some applications and systems. Multi-core architectures are being developed, but so are the alternatives. An especially strong contender for established markets is to integrate more peripheral functions into the chip.

3.5 Advantages of Dual Core Processor

- Proximity of multiple CPU cores on the same die have the advantage that the cache coherency circuitry can operate at a much higher clock rate than is possible if the signals have to travel off-chip, so combining equivalent CPUs on a single die significantly improves the performance of cache snoop operations.
- Assuming that the die can fit into the package, physically, the multi-core CPU designs require much less Printed Circuit Board (PCB) space than multi-chip SMP designs.
- A dual-core processor uses slightly less power than two coupled single-core processors, principally because of the increased power required to drive signals

external to the chip and because the smaller silicon process geometry allows the cores to operate at lower voltages. As such latency is reduced which enables a faster through-put.

- In terms of competing technologies for the available silicon die area, multi-core design can make use of proven CPU core library designs and produce a product with lower risk of design error than devising a new wider core design. Also, adding more cache suffers from diminishing returns.

3.6 Disadvantages

- Multi-core processors require operating system support to make optimal use of the second computing resource. Also, making optimal use of multiprocessing in a desktop context requires application software support.
- The higher integration of the multi-core chip drives the production yields down and are more difficult to manage thermally than lower density single-chip designs.
- From an architectural point of view, ultimately, single CPU designs may make better use of the silicon surface area than multiprocessing cores, so a development commitment to this architecture may carry the risk of obsolescence.
- Scaling efficiency is largely dependent on the application or problem set. For example, applications that require processing large amounts of data with low computer-overhead algorithms may find this architecture has an I/O bottleneck, underutilizing the device, and overall migration to and from the threading can also be effected by this.

3.7 Software Development Considerations

Assuming the operating system is appropriate for the hardware system, software that runs on dual processors systems should run on HT Technology capable/enabled processors and on dual core processor systems without modification. Even if the software

is not multi-threaded, it can still take advantage of multiple physical and/or logical processors in a multi-tasking environment. For example, a software developer could answer email or research a technical problem on the internet while a large software application is being compiled in the background. Although all applications should run on multi-processor systems, multi-threaded applications should benefit the most from the multi-processor systems discussed above. In order to get the best performance, it may be necessary to tune or optimize the application to take advantage of a specific architecture or multi-processor implementation.

3.8 Software Impact

Most existing software is not ready to directly utilize the power of multicore processors since they are written in traditional sequential programming languages like C, C++ and FORTRAN, all of which have the limited scope of only one processor in mind. Parallel programming is a must option for a single software to exploit multiple computation units(cores) simultaneously, often by multithread or multitask programming. Some existing parallel programming models such as OpenMP and MPI can be directly used on multi-core platforms. Other research efforts have been seen also, like Cray's Chapel, Sun's Fortress, and IBM's X10.

3.9 Licensing

Another issue is the question of software licensing for multi-core CPUs. Typically enterprise server software is licensed "per processor". In the past a CPU was a processor and there was no ambiguity. Now there is the possibility of counting cores as processors and charging a customer for two licenses when they use a dual-core CPU. However, the trend seems to be counting dual-core chips as a single processor as Microsoft, Intel, and AMD support this view. Oracle counts AMD and Intel dual-core CPUs as a single processor but has other numbers for other types. IBM, HP and Microsoft count a multi-

chip-module as multiple processors. If multi-chip-modules counted as one processor then CPU makers would have an incentive to make large expensive multi-chip-modules so their customers saved on software licensing. So it seems like the industry is slowly heading towards counting each die as a processor, no matter how many cores each die has. Intel has released Paxville which is really a multi-chip-module but Intel is calling it a dual-core. It is not clear yet how licensing will work for Paxville. This is an unresolved and thorny issue for software companies and customers.

CHAPTER FOUR

SINGLE CORE SCHEDULER SOFTWARE DEVELOPMENT

4.1 Steps in Development

Number	Milestone Name	Milestone Description	Timeline	Remarks Percentage completion (approximate)
1.	Requirement Specification	1. Collecting all requirements 2. Reading the required topics	Wk-01	2.5%
2.	High Level Design	1.Detailing the design 2.List of inputs and outputs 3 Restriction / Limitations	wk -02	5%
3.	Detailed Design	1. List of design options(e.g data structures), pros and cons of each option, which option is taken and the rational behind that. 2. Design details broken functionality wise. 3. List of class and the interfaces of that class. 4. File names. (Have to follow standard naming convention)	Wk -03	7.5%
4.	Coding	Writing code according to the laid down protocols and saving and compiling	Wk - 06	15%

		at each stage.		
5.	Design Testing	Mapping of the test cases to the requirement (already mentioned in HLD), and status of each test case.	Wk- 9	2.5%
6.	Review	Review of deliverables	Wk-10	2.5%

4.2 Requirement Specification

The software should be able to schedule the tasks according to the round robin algorithm while assuming that the processor has only one execution core. The demo program built in to the scheduler software must be able to take the inputs and then accurately show the working of the scheduler and also generate two diagnostic files; one of which contains the runtime information of the scheduler. This file is called single.txt. The other must contain only the performance parameters of the scheduler. This file is called singint.txt. The complete single core operating system scheduler software file is called singlecore.exe.

4.3 High Level Design

1.3.1 Inputs

- The number of tasks to be processed.
- The probability of one task entering the system in one time quantum.
- The probability of clash, i.e. two tasks entering the system in one time quantum.
- The percentage of Input/Output bound tasks.
- The percentage of CPU bound tasks.
- The probability of interrupt request in one time quantum.

1.3.2 Outputs

- Total duration of runtime of the scheduler in seconds
- Total number of tasks which are Input/Output bound.
- Total number of tasks which are CPU bound.
- The average time that a task spends in the wait queue.
- The average time that CPU bound tasks spend in the ready queue.
- The average time that Input/Output bound tasks spend in the ready queue.
- The average time that a task spends in the ready queue.
- The percentage of time the CPU is busy.
- The average number of tasks processed per hour by the system.
- The number of interrupt requests handled by the system.

1.3.3 Restrictions

- The distribution of CPU service times for jobs must be equal to 100.
- The job class distribution must be equal to 100.
- The probability of entering tasks' can not exceed 1.
- There must be at least one task entering the system.
- The probability of interrupt request can not exceed 1

4.4 Detailed Design

1.4.1 Data Structure Design

a. Name: queue

C++ Data Type: class

Purpose: It is used to store the jobs in the order that they are to be executed.

Function: It uses a C++ data structure “queue” to store the tasks due to their execution time. Since the CPU can only execute one task at a time, this queue sends tasks to the system due to their correct time slices.

Definition: The class is defined as follows:

```
class queue{  
    int front,rear;  
    task *tasks;  
    int max_length;  
    public:  
    queue(int max);  
    ~queue();  
    int is_empty();  
    int is_full();  
    void enqueue(task new_task);  
    task dequeue(void);  
    int get_front(void);  
    int get_length();  
};
```

Conditions: There must be enough space to store $\text{max_length} * \text{sizeof}(\text{task})$.

Processing: The values are stored in the memory using c memory allocation concepts.

b. **Name:** task

C++ Data Type: structure

Purpose: It is used to store the values of the fields of a task.

Function: It uses a structure to store the tasks' properties.

Definition: The structure is defined as follows:

```
struct task {  
    int task_number;  
    double task_class;  
    int CPU_time;
```

```

        double time_entered;
        double time_waited;
        double time_processed;
    };

```

Conditions: There must be enough space to create a task structure.

Processing: The values are stored in the memory using c memory allocation concepts.

c. Name: task_profiles

C++ Data Type: structure

Purpose: It is used to store the input that the user enters.

Function: It uses a structure to store user's inputs.

Definition: The structure is defined as follows:

```

struct task_profiles{
    int number_of_tasks;
    double p_task_one;
    double p_task_two;
    double IO;
    double CPU;
    double prob_of_interrupt_req;
};

```

Conditions: There must be enough space to create a parameters structure.

Processing: The values are stored in the memory using c memory allocation concepts.

d. Name: job

C++ Data Type: structure

Purpose: It is used to store the tasks in the arrays when they are sorted. (So their properties are gotten correctly and random assignments are made.)

Function: It uses a structure to store tasks' properties.

Definition: The structure is defined as follows:

```
struct job{  
    double weight;  
    double type;  
};
```

Conditions: There must be enough space to create a element structure.

Processing: The values are stored in the memory using c memory allocation concepts.

1.4.2 File Name Standards

- The source file of the single core scheduler software is called singlecore.cpp.
- The executable file of the single core scheduler software is called singlecore.exe.
- The diagnostic file generated by the single core scheduler software is called single.txt.
- The file containing the performance parameters of the single core scheduler generated by the software is called singleint.txt.

4.5 Coding

The code snippet of the main part of the software is given below:

```
if(!wait_queue.is_empty())  
{  
    waiting=wait_queue.dequeue();  
    waiting_task_number=waiting.task_number;  
    waiting.time_waited+=1.0;  
    wait_queue.enqueue(waiting);
```

```

}
while(!wait_queue.is_empty())
{
    if(wait_queue.get_front()!=waiting_task_number)
    {
        waiting=wait_queue.dequeue();
        waiting.time_waited+=1.0;
        wait_queue.enqueue(waiting);
    }
    else break;
}

for(double stime=0.0;stime<=1.0;stime+=0.1)
{
    if(!(Ready_queue.is_empty()))
    {
        interrupt_req=(rand()%1000)/1000.0;
        out<<"\n\n\n-----
        interrupt_req="<<interrupt_req<<"-----
        prof.prob_of_interrupt_req="<<prof.prob_of_interrupt_req<<"-----
        -----no_of_interrupt_reqs="<<no_of_interrupt_reqs;
        if(interrupt_req>prof.prob_of_interrupt_req)
        {
            dequeued=Ready_queue.dequeue();
            if(dequeued.task_class==0.2)
            stime+=0.1;
            dequeued.time_processed+=dequeued.task_class;
            if(dequeued.CPU_time>dequeued.time_processed)

```

```

Ready_queue.enqueue(dequeued);
else{
    if(dequeued.task_class==0.1)
        execution_time_of_IO_bound_tasks+=clock-
dequeued.time_waited-1+stime-
dequeued.time_entered;

    else if(dequeued.task_class==0.2)
        execution_time_of_CPU_bound_tasks+=clock-
dequeued.time_waited-1+stime-
dequeued.time_entered;

    execution_time_of_tasks+=clock-
dequeued.time_waited-1+stime-
dequeued.time_entered;
}
}
else
{
    Ready_queue.enqueue(Ready_queue.dequeue());
    no_of_interrupt_reqs++;
}
}
else idle_time++;
}

```


4.6 Software Testing

The software is tested for the following cases of inputs.

Case 1:

Inputs:

- The number of tasks to be processed: 50
- The probability of one task entering the system in one time quantum: 0.1
- The probability of clash, i.e. two tasks entering the system in one time quantum: 0.01
- The percentage of Input/Output bound tasks: 50
- The percentage of CPU bound tasks: 50
- The probability of interrupt request in one time quantum 0.001

Outputs:

- Total duration of runtime of the scheduler in seconds: 1138
- Total number of tasks which are Input/Output bound: 27
- Total number of tasks which are CPU bound: 23
- The average time that a task spends in the wait queue: 13.96
- The average time that CPU bound tasks spend in the ready queue: 210.469565
- The average time that Input/Output bound tasks spend in the ready queue: 370.833333
- The average time that a task spends in the ready queue: 297.066
- The percentage of time the CPU is busy: 99.912127
- The average number of tasks processed per hour by the system: 158.172232
- The number of interrupt requests handled by the system: 23

Result for case 1: Pass

Case 2:

Inputs:

- The number of tasks to be processed: 100
- The probability of one task entering the system in one time quantum: 0.1
- The probability of clash, i.e. two tasks entering the system in one time quantum: 0.01
- The percentage of Input/Output bound tasks: 99
- The percentage of CPU bound tasks: 1
- The probability of interrupt request in one time quantum: 0.001

Outputs:

- Total duration of runtime of the scheduler in seconds: 2664
- Total number of tasks which are Input/Output bound: 99
- Total number of tasks which are CPU bound: 1
- The average time that a task spends in the wait queue: 23.29
- The average time that CPU bound tasks spend in the ready queue: 1741
- The average time that Input/Output bound tasks spend in the ready queue: 983.813131
- The average time that a task spends in the ready queue: 991.385
- The percentage of time the CPU is busy: 99.624625
- The average number of tasks processed per hour by the system: 135.135135
- The number of interrupt requests handled by the system: 60

Result for case 2: Pass

Case 3:

Inputs:

- The number of tasks to be processed: 150
- The probability of one task entering the system in one time quantum: 0.01
- The probability of clash, i.e. two tasks entering the system in one time quantum: 0.001
- The percentage of Input/Output bound tasks: 60
- The percentage of CPU bound tasks: 40
- The probability of interrupt request in one time quantum: 0.0001

Outputs:

- Total duration of runtime of the scheduler in seconds: 9536
- Total number of tasks which are Input/Output bound: 90
- Total number of tasks which are CPU bound: 60
- The average time that a task spends in the wait queue: 0
- The average time that CPU bound tasks spend in the ready queue: 36.865
- The average time that Input/Output bound tasks spend in the ready queue: 51.305556
- The average time that a task spends in the ready queue: 45.529333
- The percentage of time the CPU is busy: 42.512584
- The average number of tasks processed per hour by the system: 56.627517
- The number of interrupt requests handled by the system: 43

Result for case 3: Pass

Case 4:

Inputs:

- The number of tasks to be processed: 200
- The probability of one task entering the system in one time quantum: 0.4
- The probability of clash, i.e. two tasks entering the system in one time quantum: 0.6
- The percentage of Input/Output bound tasks: 50
- The percentage of CPU bound tasks: 50
- The probability of interrupt request in one time quantum: 0.01

Outputs:

- Total duration of runtime of the scheduler in seconds: 5632
- Total number of tasks which are Input/Output bound: 104
- Total number of tasks which are CPU bound: 97
- The average time that a task spends in the wait queue: 26.725
- The average time that CPU bound tasks spend in the ready queue: 2594.896907
- The average time that Input/Output bound tasks spend in the ready queue: 3198.966346
- The average time that a task spends in the ready queue: 2921.9875
- The percentage of time the CPU is busy: 100
- The average number of tasks processed per hour by the system: 127.840909
- The number of interrupt requests handled by the system: 5125

Result for case 4: Pass

Case 5:

Inputs:

- The number of tasks to be processed: 250
- The probability of one task entering the system in one time quantum: 0.02
- The probability of clash, i.e. two tasks entering the system in one time quantum: 0.001
- The percentage of Input/Output bound tasks: 50
- The percentage of CPU bound tasks: 50
- The probability of interrupt request in one time quantum: 0.001

Outputs:

- Total duration of runtime of the scheduler in seconds: 11322
- Total number of tasks which are Input/Output bound: 133
- Total number of tasks which are CPU bound: 117
- The average time that a task spends in the wait queue: 0
- The average time that CPU bound tasks spend in the ready queue: 48.553846
- The average time that Input/Output bound tasks spend in the ready queue: 64.709023
- The average time that a task spends in the ready queue: 57.1484
- The percentage of time the CPU is busy: 56.20915
- The average number of tasks processed per hour by the system: 79.491256
- The number of interrupt requests handled by the system: 118

Result for case 5: Pass

4.7 Software Pitfalls to Avoid

4.7.1 Deadlock Avoidance

In the computing world **deadlock** refers to a specific condition when two or more processes are each waiting for another to release a resource, or more than two processes are waiting for resources in a circular chain. Deadlocks are a common problem in multiprocessing where many processes share a specific type of mutually exclusive resource known as a *software*, or *soft*, lock.

Example:

```
Lock *l1, *l2;
```

```
void p() {
```

```
    l1->Acquire();
```

```
    l2->Acquire();//code manipulates data that l1 and l2 protect
```

```
    l2->Release();
```

```
    l1->Release();
```

```
}
```

```
void q() {
```

```
    l2->Acquire();
```

```
    l1->Acquire();//code manipulates data that l1 and l2 protect
```

```
    l1->Release();
```

```
    l2->Release();
```

```
}
```

If p and q execute concurrently, consider what may happen. First, p acquires l1 and q acquires l2. Then, p waits to acquire l2 and q waits to acquire l1. This case is called deadlock

4.7.2 Avoidance Of Creation of Dangling Pointers

Since the program requires in depth Data Structures concepts, so one should be very careful as to how to avoid the creation of pointers that is no longer allocated. Dangling pointers are nasty bugs because they seldom crash the program until long after they have been created, which makes them hard to find. Programs that create dangling pointers often appear to work on small inputs, but are likely to fail on large or complex inputs.

```
delete [] s1;
delete [] s2;
return f(s1, s2); // s1 and s2 are dangling pointers
```

4.7.3 Avoidance Of Memory Leak

A memory leak is a particular kind of unnecessary memory consumption by a computer program, where the program fails to release memory that is no longer needed.

A memory leak can diminish the performance of the computer by reducing the amount of available memory. Memory allocation is normally a component of the operating system, so the result of a memory leak is usually an ever growing amount of memory being used by the system as a whole, not merely by the erroneous process or program. Eventually, in the worst case, too much of the available memory may become allocated and all or part of the system or device stops working correctly or the application fails.

4.7.4 Unreachable Memory in C++

Consider the following example and notice that memory was available and pointed to by *s*, but not saved. After this function returns, the pointer is destroyed and the allocated memory becomes unreachable to "fix" this code,

you would add the statement "free(s)" to the else block before the "return 0" statement.

```
int f(void)
{
    char* s;
    s = malloc(50); /* get memory */
    if (s==NULL)
        return 1; /* no memory available */
    else
    { /* memory available */
        return 0; /* memory leak - see note below */
    }
}

int main(void)
{
    /* this is an infinite loop calling the above function */
    while (1)
        f(); /* This function call will fail to malloc sooner or later */
    return 0;
}
```

4.7.5 Keeping Track of Time Elapsed

Time is usually maintained in a counter. Size of counters should be decided such that it can take the whole range of values after increments. Counters should be adjusted in such a way that it is very accurate, neither a cycle nor a cycle more. Nowhere in the program should the counters be over-ridden.

1.8 Algorithm / Implementation

1. The probability of entering tasks is sorted.
2. The cpu time requirements are sorted.
3. The task classes are sorted.
4. While first process enters the wait queue
 - 4.1 Random number is generated
5. The task number ++
6. Determine process category using random number
7. Determine total processing time using random number
8. Make time spent in wait queue =0
9. Make time processed = 0.0
10. Enqueue in the wait queue.
11. Generate total processes required for the demo
12. While the Wait queue is not empty and ready queue is not full
 - 12.1 Dequeue first process from the wait queue and enqueue in the cpu queue while the wait queue is not empty and the cpu queue is not full.
 - 12.2 Increment the positions of remaining processes in wait queue by 1.
 - 12.3 If interrupt requested then jump to 12.5
 - 12.4 Increment the time processed of first process in ready queue by 0.1.
 - 12.5 If (CPU processed time < total processing time required)
 - Enqueue the first process at the back of ready queue and update other processes.
 - Else
 - Dequeue the first process of ready queue and enqueue the first process of wait queue into the ready queue.
13. Stop.

4.9 Research Instruments / Tools

The given algorithm is being implemented on a computer with the following configuration:

- C.P.U : AMD Athlon 64-bit Processor 2800+
- Clock Speed of C.P.U. : 2.0 GHz
- Operating System : Microsoft ® Windows Version 5.1

(Build 2600.xpsp_sp2_rtm.040803-2158 : Service Pack 2)

Copyright © 1981-2001 Microsoft Corporation

- C++ Compiler : Turbo C++ Version 3.0

Copyright © 1991, 1995 Borland International, Inc.

4.10 Snapshots of the Software Interface

```
C:\ATC\BGVZZPROJ-2.EXE
#####
#-----Operating System-----#
#-----Scheduler-----#
#####
Press any key to continue . . .
The number of tasks to be processed is...
100
The probability of 1 task entering the system in one second is 0.03
The probability of 2 tasks entering the system at the same time in one second is
0.001
The task class distribution is...
Percentage of I/O bound class is...35
Percentage of CPU bound class is...65
Enter the probability of interrupt request...0.004

Task_profiles are gathered successfully!
Press any key to continue . . .

Demo is finished...
See the results from the "OSched3.txt" file...
```

Figure 5. Snapshot of Main Application Window

```
OSSCHEDS - Notepad
File Edit Format View Help
DEMO ENDS after 6497 seconds...

Parameters---->
Number of tasks which are I/O bound...127
Number of tasks which are CPU bound...123

Number of tasks ...:250

The average time that a task spends in the wait queue...:21.436

The average time that CPU Bound tasks spend in the Ready queue ...:567.899187
The average time that IO Bound tasks spend in the Ready queue ...:676.372441
The average time that a task spent in the Ready queue ...:623.0036

CPU is busy for the 99.599815 % of the simulation time..

The average number of tasks processed per hour by the system...:138.525473

The number of interrupt requestsby the system...:205
```

Figure 2. Snapshot of output generated by the software

CHAPTER FIVE

DUAL CORE SCHEDULER SOFTWARE DEVELOPMENT

5.1 Steps in Development

Number	Milestone Name	Milestone Description	Timeline	Remarks Percentage completion (approximate)
1.	Requirement Specification	1. Collecting all requirements 2. Reading the required topics	Wk-11	2.5%
2.	High Level Design	1.Detailing the design 2.List of inputs and outputs 3 Restriction / Limitations	wk -12	5%
3.	Detailed Design	1. List of design options(e.g. data structures), pros and cons of each option, which option is taken and the rational behind that. 2. Design details broken functionality wise. 3. List of class and the interfaces of that class. 4. File names. (Have to follow standard naming convention)	Wk -13	7.5%
4.	Coding	Writing code according to the laid down protocols and saving and compiling	Wk - 15	15%

		at each stage.		
5.	Design Testing	Mapping of the test cases to the requirement (already mentioned in HLD), and status of each test case.	Wk- 16	2.5%
6.	Review	Review of deliverables	Wk-17	2.5%

5.2 Requirement Specification

The software should be able to schedule the tasks according to the algorithm developed by the authors while assuming that the processor has two execution cores. The demo program built in to the scheduler software must be able to take the inputs and then accurately show the working of the scheduler and also generate two diagnostic files; one of which contains the runtime information of the scheduler. This file is called dual.txt. The other must contain only the performance parameters of the scheduler. This file is called dualint.txt. The complete single core operating system scheduler software file is called dualcore.exe.

5.3 High Level Design

1.3.1 Inputs

- The number of tasks to be processed.
- The probability of one task entering the system in one time quantum.
- The probability of clash, i.e. two tasks entering the system in one time quantum.
- The percentage of Input/Output bound tasks.
- The percentage of CPU bound tasks.
- The probability of interrupt request in one time quantum.

1.3.2 Outputs

- Total duration of runtime of the scheduler in seconds
- Total number of tasks which are Input/Output bound.
- Total number of tasks which are CPU bound.
- The average time that a task spends in the wait queue.
- The average time that CPU bound tasks spend in the ready queue.
- The average time that Input/Output bound tasks spend in the ready queue.
- The average time that a task spends in the ready queue.
- The percentage of time Core 0 is busy.
- The percentage of time Core 1 is busy.
- The average number of tasks processed per hour by the system.
- The number of interrupt requests handled by the system.

1.3.3 Restrictions

- The distribution of CPU service times for jobs must be equal to 100.
- The job class distribution must be equal to 100.
- The probability of entering tasks' can not exceed 1.
- There must be at least one task entering the system.
- The probability of interrupt request can not exceed 1.

5.4 Detailed Design

1.4.1 Data Structure Design

- e. Name: queue
C++ Data Type: class

Purpose: It is used to store the jobs in the order that they are to be executed.

Function: It uses a C++ data structure "queue" to store the tasks due to their execution time. Since the CPU can only execute one task at a time, this queue sends tasks to the system due to their correct time slices.

Definition: The class is defined as follows:

```
class queue{
    int front,rear;
    task *tasks;
    int max_length;
public:
    queue(int max);
    ~queue();
    int is_empty();
    int is_full();
    void enqueue(task new_task);
    task dequeue(void);
    int get_front(void);
    int get_length();
};
```

Conditions: There must be enough space to store $\text{max_length} * \text{sizeof}(\text{task})$.

Processing: The values are stored in the memory using c memory allocation concepts.

f. **Name:** task

C++ Data Type: structure

Purpose: It is used to store the values of the fields of a task.

Function: It uses a structure to store the tasks' properties.

Definition: The structure is defined as follows:

```
struct task{
    int task_number;
```



```
double task_class;  
int CPU_time;  
double time_entered;  
double time_waited;  
double time_processed;  
};
```

Conditions: There must be enough space to create a task structure.

Processing: The values are stored in the memory using c memory allocation concepts.

g. Name: task_profiles

C++ Data Type: structure

Purpose: It is used to store the input that the user enters.

Function: It uses a structure to store user's inputs.

Definition: The structure is defined as follows:

```
struct task_profiles {  
    int number_of_tasks;  
    double p_task_one;  
    double p_task_two;  
    double IO;  
    double CPU;  
    double prob_of_interrupt_req;  
};
```

Conditions: There must be enough space to create a parameters structure.

Processing: The values are stored in the memory using c memory allocation concepts.

h. Name: job

C++ Data Type: structure

Purpose: It is used to store the tasks in the arrays when they are sorted. (So their properties are gotten correctly and random assignments are made.)

Function: It uses a structure to store tasks' properties.

Definition: The structure is defined as follows:

```
struct job{  
    double weight;  
    double type;  
};
```

Conditions: There must be enough space to create a element structure.

Processing: The values are stored in the memory using c memory allocation concepts.

1.4.2 File Name Standards

- The source file of the single core scheduler software is called dualcore.cpp.
- The executable file of the single core scheduler software is called dualcore.exe.
- The diagnostic file generated by the single core scheduler software is called dual.txt.
- The file containing the performance parameters of the single core scheduler generated by the software is called dualint.txt.

5.5 Coding

The code snippet of the main part of the software is given below:

```
for(double  
short_time_0=0.0,short_time_1=0.0;(short_time_0<=1.0)&&(short_time_1<=1.0)  
;short_time_0+=0.1,short_time_1+=0.1)
```

```

{
  if(!(Ready_queue.is_empty()))&&(short_time_0!=1.0)
  {
    interrupt_req=(rand()%1000)/1000.0;
    //out<<"\n\n\n-----
    interrupt_req="<<interrupt_req<<"-----
    prof.prob_of_interrupt_req="<<prof.prob_of_interrupt_req<<"-----
    -----no_of_interrupt_reqs="<<no_of_interrupt_reqs;
    if(interrupt_req>prof.prob_of_interrupt_req)
    {
      dequeued=Ready_queue.dequeue();
      if(dequeued.task_class==0.2)
        short_time_0+=0.1;
      dequeued.time_processed+=dequeued.task_class;
      if(dequeued.CPU_time>dequeued.time_processed)
        Ready_queue.enqueue(dequeued);
      else{
        if(dequeued.task_class==0.1)
          execution_time_of_IO_bound_tasks+=clock-
          dequeued.time_waited-1+short_time_0-
          dequeued.time_entered;

        else if(dequeued.task_class==0.2)
          execution_time_of_CPU_bound_tasks+=clock-
          dequeued.time_waited-1+short_time_0-
          dequeued.time_entered;

        execution_time_of_tasks+=clock-
        dequeued.time_waited-1+short_time_0-
        dequeued.time_entered;
      }
    }
  }
}

```

```

    }
    else
    {
        Ready_queue.enqueue(Ready_queue.dequeue());
        no_of_interrupt_reqs++;
    }
}
else idle_time_0++;

if(!(Ready_queue.is_empty()))&&(short_time_1!=1.0)
{
    interrupt_req=(rand()%1000)/1000.0;
    //out<<"\n\n\n-----
    interrupt_req="<<interrupt_req<<"-----
    prof.prob_of_interrupt_req="<<prof.prob_of_interrupt_req<<"-----
    -----no_of_interrupt_reqs="<<no_of_interrupt_reqs;
    if(interrupt_req>prof.prob_of_interrupt_req)
    {
        dequeued=Ready_queue.dequeue();
        if(dequeued.task_class==0.2)
            short_time_1+=0.1;
        dequeued.time_processed+=dequeued.task_class;
        if(dequeued.CPU_time>dequeued.time_processed)
            Ready_queue.enqueue(dequeued);
        else{
            if(dequeued.task_class==0.1)
                execution_time_of_IO_bound_tasks+=clock-
                dequeued.time_waited-1+short_time_1-
                dequeued.time_entered;

            else if(dequeued.task_class==0.2)

```

```
execution_time_of_CPU_bound_tasks+=clock-  
dequeued.time_waited-1+short_time_1-  
dequeued.time_entered;
```

```
execution_time_of_tasks+=clock-dequeued.time_waited-  
1+short_time_1-dequeued.time_entered;  
}
```

```
}  
else
```

```
{
```

```
Ready_queue.enqueue(Ready_queue.dequeue());  
no_of_interrupt_reqs++;
```

```
}
```

```
}  
else idle_time_1++;
```

```
}
```

5.6 Software Testing

The software is tested for the following cases of inputs.

Case 1:

Inputs:

- The number of tasks to be processed: 50
- The probability of one task entering the system in one time quantum: 0.1
- The probability of clash, i.e. two tasks entering the system in one time quantum: 0.01
- The percentage of Input/Output bound tasks: 50
- The percentage of CPU bound tasks: 50
- The probability of interrupt request in one time quantum 0.001

Outputs:

- Total duration of runtime of the scheduler in seconds: 691
- Total number of tasks which are Input/Output bound: 22
- Total number of tasks which are CPU bound: 28
- The average time that a task spends in the wait queue: 8.44
- The average time that CPU bound tasks spend in the ready queue: 165.396429
- The average time that Input/Output bound tasks spend in the ready queue:
211.531818
- The average time that a task spends in the ready queue: 185.696
- The percentage of time the Core 0 is busy: 99.855282
- The percentage of time the Core 1 is busy: 99.855282
- The average number of tasks processed per hour by the system: 260.492041
- The number of interrupt requests handled by the system: 19

Result for case 1: Pass

Case 2:

Inputs:

- The number of tasks to be processed: 100
- The probability of one task entering the system in one time quantum: 0.1
- The probability of clash, i.e. two tasks entering the system in one time quantum: 0.01
- The percentage of Input/Output bound tasks: 99
- The percentage of CPU bound tasks: 1
- The probability of interrupt request in one time quantum: 0.001

Outputs:

- Total duration of runtime of the scheduler in seconds: 1105
- Total number of tasks which are Input/Output bound: 100
- Total number of tasks which are CPU bound: 0
- The average time that a task spends in the wait queue: 6.58
- The average time that CPU bound tasks spend in the ready queue: 0
- The average time that Input/Output bound tasks spend in the ready queue: 147.603
- The average time that a task spends in the ready queue: 147.603
- The percentage of time the Core 0 is busy: 97.013575
- The percentage of time the Core 1 is busy: 97.013575
- The average number of tasks processed per hour by the system: 325.791855
- The number of interrupt requests handled by the system: 35

Result for case 2: Pass

Case 3:

Inputs:

- The number of tasks to be processed: 150
- The probability of one task entering the system in one time quantum: 0.01
- The probability of clash, i.e. two tasks entering the system in one time quantum: 0.001
- The percentage of Input/Output bound tasks: 60
- The percentage of CPU bound tasks: 40
- The probability of interrupt request in one time quantum: 0.0001

Outputs:

- Total duration of runtime of the scheduler in seconds: 12182
- Total number of tasks which are Input/Output bound: 97
- Total number of tasks which are CPU bound: 53
- The average time that a task spends in the wait queue: 0
- The average time that CPU bound tasks spend in the ready queue: 15.566038
- The average time that Input/Output bound tasks spend in the ready queue: 19.31134
- The average time that a task spends in the ready queue: 17.988
- The percentage of time the Core 0 is busy: 16.532589
- The percentage of time the Core 1 is busy: 16.532589
- The average number of tasks processed per hour by the system: 44.327697
- The number of interrupt requests handled by the system: 36

Result for case 3: Pass

Case 4:

Inputs:

- The number of tasks to be processed: 200
- The probability of one task entering the system in one time quantum: 0.4
- The probability of clash, i.e. two tasks entering the system in one time quantum: 0.6
- The percentage of Input/Output bound tasks: 50
- The percentage of CPU bound tasks: 50
- The probability of interrupt request in one time quantum: 0.01

Outputs:

- Total duration of runtime of the scheduler in seconds: 2712
- Total number of tasks which are Input/Output bound: 102
- Total number of tasks which are CPU bound: 98
- The average time that a task spends in the wait queue: 12.94
- The average time that CPU bound tasks spend in the ready queue:
1198.902041
- The average time that Input/Output bound tasks spend in the ready queue:
1404.166667
- The average time that a task spends in the ready queue: 1303.587
- The percentage of time the Core 0 is busy: 100
- The percentage of time the Core 1 is busy: 100
- The average number of tasks processed per hour by the system: 265.486726
- The number of interrupt requests handled by the system: 480

Result for case 4: Pass

Case 5:

Inputs:

- The number of tasks to be processed: 250
- The probability of one task entering the system in one time quantum: 0.02
- The probability of clash, i.e. two tasks entering the system in one time quantum: 0.001
- The percentage of Input/Output bound tasks: 50
- The percentage of CPU bound tasks: 50
- The probability of interrupt request in one time quantum: 0.001

Outputs:

- Total duration of runtime of the scheduler in seconds: 10402
- Total number of tasks which are Input/Output bound: 128
- Total number of tasks which are CPU bound: 122
- The average time that a task spends in the wait queue: 0
- The average time that CPU bound tasks spend in the ready queue: 16.578689
- The average time that Input/Output bound tasks spend in the ready queue: 18.839062
- The average time that a task spends in the ready queue: 17.736
- The percentage of time the Core 0 is busy: 30.830609
- The percentage of time the Core 1 is busy: 30.830609
- The average number of tasks processed per hour by the system: 86.521823
- The number of interrupt requests handled by the system: 111

Result for case 5: Pass

5.7 Software Pitfalls to Avoid

5.7.1 Deadlock Avoidance

In the computing world **deadlock** refers to a specific condition when two or more processes are each waiting for another to release a resource, or more than two processes are waiting for resources in a circular chain. Deadlocks are a common problem in multiprocessing where many processes share a specific type of mutually exclusive resource known as a *software*, or *soft*, lock.

Example:

```
Lock *l1, *l2;
```

```
void p() {  
    l1->Acquire();  
    l2->Acquire();//code manipulates data that l1 and l2 protect
```

```
    l2->Release();  
    l1->Release();  
}
```

```
void q() {  
    l2->Acquire();  
    l1->Acquire();//code manipulates data that l1 and l2 protect  
    l1->Release();  
    l2->Release();  
}
```

If p and q execute concurrently, consider what may happen. First, p acquires l1 and q acquires l2. Then, p waits to acquire l2 and q waits to acquire l1. This case is called deadlock

5.7.2 *Avoidance Of Creation of Dangling Pointers*

Since the program requires in depth Data Structures concepts, so one should be very careful as to how to avoid the creation of pointers that is no longer allocated. Dangling pointers are nasty bugs because they seldom crash the program until long after they have been created, which makes them hard to find. Programs that create dangling pointers often appear to work on small inputs, but are likely to fail on large or complex inputs.

```
delete [] s1;
delete [] s2;
return f(s1, s2); // s1 and s2 are dangling pointers
```

5.7.3 *Avoidance Of Memory Leak*

A memory leak is a particular kind of unnecessary memory consumption by a computer program, where the program fails to release memory that is no longer needed.

A memory leak can diminish the performance of the computer by reducing the amount of available memory. Memory allocation is normally a component of the operating system, so the result of a memory leak is usually an ever growing amount of memory being used by the system as a whole, not merely by the erroneous process or program. Eventually, in the worst case, too much of the available memory may become allocated and all or part of the system or device stops working correctly or the application fails.

5.7.4 *Unreachable Memory in C++*

Consider the following example and notice that memory was available and pointed to by *s*, but not saved. After this function returns, the pointer is destroyed and the allocated memory becomes unreachable to "fix" this code,

you would add the statement "free(s)" to the else block before the "return 0" statement.

```
int f(void)
{
    char* s;
    s = malloc(50); /* get memory */
    if (s==NULL)
        return 1; /* no memory available */
    else
    { /* memory available */
        return 0; /* memory leak - see note below */
    }
}

int main(void)
{
    /* this is an infinite loop calling the above function */
    while (1)
        f(); /* This function call will fail to malloc sooner or later */
    return 0;
}
```

5.7.5 *Keeping Track of Time Elapsed*

Time is usually maintained in a counter. Size of counters should be decided such that it can take the whole range of values after increments. Counters should be adjusted in such a way that it is very accurate, neither a cycle nor a cycle more. Nowhere in the program should the counters be over-ridden.

5.8 Algorithm / Implementation

- 1 The probability of entering tasks is sorted.
- 2 The cpu time requirements are sorted.
- 3 The task classes are sorted.
- 4 While first process enters the wait queue
 - 4.1 Random number is generated
- 5 The task number ++
- 6 Determine process category using random number
- 7 Determine total processing time using random number
- 8 Make time spent in wait queue =0
- 9 Make time processed = 0.0
- 10 Enqueue in the wait queue.
- 11 Generate total processes required for the demo
- 12 While the Wait queue is not empty and ready queue is not full
 - 12.1 Dequeue first process from the wait queue and enqueue in the cpu queue while the wait queue is not empty and the cpu queue is not full.
 - 12.2 Increment the positions of remaining processes in wait queue by 1.
 - 12.3 If interrupt requested then jump to 12.5
 - 12.4 Increment the time processed of first process in ready queue by 0.1.
 - 12.5 If (CPU processed time < total processing time required)
Enqueue the first process at the back of ready queue and update other processes.
Else
Dequeue the first process of ready queue and enqueue the first process of wait queue into the ready queue.
 - 12.6 Increment the time processed of first process in ready queue by 0.1.
 - 12.7 If (CPU processed time < total processing time required)
Enqueue the first process at the back of ready queue and update other processes.
Else

Dequeue the first process of ready queue and enqueue the first process of wait queue into the ready queue.

13. Stop.

5.9 Research Instruments / Tools

The given algorithm is being implemented on a computer with the following configuration:

- C.P.U : AMD Athlon 64-bit Processor 2800+
- Clock Speed of C.P.U. : 2.0 GHz
- Operating System : Microsoft ® Windows Version 5.1

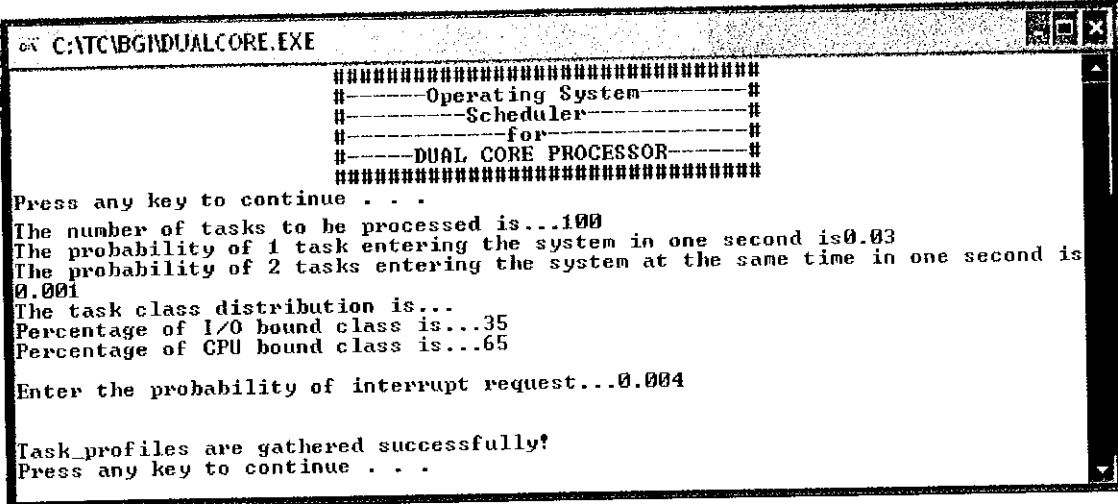
(Build 2600.xpsp_sp2_rtm.040803-2158 : Service Pack 2)

Copyright © 1981-2001 Microsoft Corporation

- C++ Compiler : Turbo C++ Version 3.0

Copyright © 1991, 1995 Borland International, Inc.

5.10 Snapshots of the Software Interface



```
C:\ATC\BGINDUALCORE.EXE
#####
#-----Operating System-----#
#-----Scheduler-----#
#-----for-----#
#-----DUAL CORE PROCESSOR-----#
#####
Press any key to continue . . .
The number of tasks to be processed is...100
The probability of 1 task entering the system in one second is0.03
The probability of 2 tasks entering the system at the same time in one second is
0.001
The task class distribution is...
Percentage of I/O bound class is...35
Percentage of CPU bound class is...65
Enter the probability of interrupt request...0.004

Task_profiles are gathered successfully!
Press any key to continue . . .
```

Figure 7. Snapshot of Main Application Window


```
DUAL - Notepad
File Edit Format View Help

DEMO ENDS after 3245 seconds...

Parameters---->
Number of tasks which are I/O bound...:40
Number of tasks which are CPU bound...:60

Number of tasks ...:100

The average time that a task spends in the wait queue...:0

The average time that CPU Bound tasks spend in the Ready queue ...:20.641667
The average time that IO Bound tasks spend in the Ready queue ...:29.08
The average time that a task spent in the Ready queue ...:24.017

Core 0 is busy for the 39.6302 % of the simulation time..
Core 1 is busy for the 39.6302 % of the simulation time..

The average number of tasks processed per hour by the system...:110.939908

The number of interrupt requestsby the system...:105
```

Figure 4. Snapshot of output generated by the software

CHAPTER SIX

GRAPHICAL UTILITY SOFTWARE DEVELOPMENT

6.1 Steps in Development

Number	Milestone Name	Milestone Description	Timeline	Remarks Percentage completion (approximate)
1.	Requirement Specification & Background Reading	1. Collecting all requirements 2. Reading the required topics	Wk-18	2.5%
2.	High Level Design	1. Detailing the design 2. List of inputs and outputs 3. Restriction / Limitations	wk -18	5%
3.	Detailed Design	1. List of design options(e.g data structures), pros and cons of each option, which option is taken and the rational behind that. 2. Design details broken functionality wise. 3. List of class and the interfaces of that class. 4. File names. (Have to follow standard naming convention)	Wk -18	7.5%
4.	Coding	Writing code according to the laid down protocols and saving and compiling	Wk - 19	10%

		at each stage.		
5.	Design Testing	Mapping of the test cases to the requirement (already mentioned in HLD), and status of each test case.	Wk- 20	2.5%
6.	Review	Review of deliverables	Wk-20	2.5%

6.2 Requirement Specification

The software should be able to generate graphs to compare the performance of the schedulers. It must be able to retrieve the inputs and then accurately show the bars of the respective parameters of performance of each scheduler and print them on screen side by side. The complete graphical utility software file is called fileread.exe.

6.3 High Level Design

1.3.1 Inputs

The two files containing only the parameters of performance of the schedulers in double c++ data type only.

1.3.2 Outputs

Graphs with the following titles:

- I/O bound tasks in Single Core Vs. Dual Core
- CPU bound tasks in Single Core Vs Dual Core
- Total tasks in Single Core Vs Dual Core
- Average time in wait queue in Single Core Vs Dual Core
- Average time in ready queue by CPU tasks in Single Vs Dual Core

- Average time in ready queue by I/O tasks in Single Vs Dual Core
- Average time in ready queue in Single Core Vs Dual Core
- Percentage of busy time of CPU in Single Core Vs Dual Core
- Average no of tasks processed/hour in Single Core Vs Dual Core
- Total interrupts in Single Core Vs Dual Core

1.3.3 Restrictions

- The files should exist in the same working directory as the software
- The BGI files of Turbo C++ compiler should exist in the same working directory as the software.

6.4 Detailed Design

1.4.1 Functions Used

- void Initialize(void);
- void Bar(double x, double y, char *p);
- void heading(void);
- void Pause(void);
- void MainWindow(char *header);
- void StatusLine(char *msg);
- void DrawBorder(void);
- void changetextstyle(int font, int direction, int charsize);
- double maximum (double *p);

1.4.2 File Name Standards

- The file containing the performance parameters of the single core scheduler generated by the software is called singleint.txt.

- The file containing the performance parameters of the dual core scheduler generated by the software is called dualint.txt.

6.5 Coding

The code snippet of the main part of the software is given below:

```
int main()
{
    char a[10][100]={"I/O bound tasks in Single Core Vs. Dual Core", "CPU
    bound tasks in Single Core Vs Dual Core","Total tasks in Single Core Vs
    Dual Core","Avg time in wait Q in Single Core Vs Dual Core","Avg time
    in Ready Q by CPU tasks in Single Vs Dual Core","Avg time in Ready Q
    by I/O tasks in Single Vs Dual Core","Avg time in Ready Q in Single
    Core Vs Dual Core","% of busy time of CPU in Single Core Vs Dual
    Core","Avg no of tasks processed/hour in Single Core Vs Dual
    Core","Total interrupts in Single Core Vs Dual Core"};
    Initialize();
    heading();
    double match_single[11], match_dual[12];
    fstream list_single,list_dual;
    list_single.open("singleint.txt",ios::in);
    list_dual.open("dualint.txt",ios::in);
    int i=0,j=0;
    while(!list_single.eof())
    {

        cout<<"\n\n...Single\t";
```

```

        list_single>>match_single[i];
        cout<<match_single[i];
        cout<<"\n\n...Dual\t";
        list_dual>>match_dual[i];
        cout<<match_dual[i];
        i++;
    }
    for(i=0,j=0;j<11;j++,i++)
    {
        BarDemo(match_single[j],match_dual[i],a[j] );
        if(i==7)i++;
    }
    BarDemo(match_dual[7],match_dual[8], "Load Comparison of Core 0 Vs
    Core 1");
    closegraph();
    return(0);
}

```

6.6 Software Testing

The software is tested for the following cases of inputs.

Case 1:

Inputs:

- The number of tasks to be processed: 50
- The probability of one task entering the system in one time quantum: 0.1

- The probability of clash, i.e. two tasks entering the system in one time quantum: 0.01
- The percentage of Input/Output bound tasks: 50
- The percentage of CPU bound tasks: 50
- The probability of interrupt request in one time quantum 0.001

Outputs:

- Graphs are properly generated.

Result for case 1: Pass

Case 2:

Inputs:

- The number of tasks to be processed: 100
- The probability of one task entering the system in one time quantum: 0.1
- The probability of clash, i.e. two tasks entering the system in one time quantum: 0.01
- The percentage of Input/Output bound tasks: 99
- The percentage of CPU bound tasks: 1
- The probability of interrupt request in one time quantum: 0.001

Outputs:

- Graphs are properly generated.

Result for case 2: Pass

Case 3:

Inputs:

- The number of tasks to be processed: 150
- The probability of one task entering the system in one time quantum: 0.01
- The probability of clash, i.e. two tasks entering the system in one time quantum: 0.001
- The percentage of Input/Output bound tasks: 60
- The percentage of CPU bound tasks: 40
- The probability of interrupt request in one time quantum: 0.0001

Outputs:

- Graphs are properly generated.

Result for case 3: Pass

Case 4:

Inputs:

- The number of tasks to be processed: 200
- The probability of one task entering the system in one time quantum: 0.4
- The probability of clash, i.e. two tasks entering the system in one time quantum: 0.6
- The percentage of Input/Output bound tasks: 50
- The percentage of CPU bound tasks: 50
- The probability of interrupt request in one time quantum: 0.01

Outputs:

- Graphs are properly generated.

Result for case 4: Pass

Case 5:

Inputs:

- The number of tasks to be processed: 250
- The probability of one task entering the system in one time quantum: 0.02
- The probability of clash, i.e. two tasks entering the system in one time quantum: 0.001
- The percentage of Input/Output bound tasks: 50
- The percentage of CPU bound tasks: 50
- The probability of interrupt request in one time quantum: 0.001

Outputs:

- Graphs are properly generated.

Result for case 5: Pass

6.7 Software Pitfalls to Avoid

6.7.1 *Deadlock Avoidance*

In the computing world deadlock refers to a specific condition when two or more processes are each waiting for another to release a resource, or more than two processes are waiting for resources in a circular chain. Deadlocks are a common problem in multiprocessing where many processes share a specific type of mutually exclusive resource known as a *software*, or *soft*, lock.

Example:

```
Lock *l1, *l2;
```

```
void p() {
```

```
    l1->Acquire();
```

```
    l2->Acquire();//code manipulates data that l1 and l2 protect
```

```
    l2->Release();
```

```

    l1->Release();
        }
void q() {
    l2->Acquire();
    l1->Acquire();//code manipulates data that l1 and l2 protect
    l1->Release();
    l2->Release();
        }

```

If p and q execute concurrently, consider what may happen. First, p acquires l1 and q acquires l2. Then, p waits to acquire l2 and q waits to acquire l1. This case is called deadlock

6.7.2 *Avoidance Of Creation of Dangling Pointers*

Since the program requires in depth Data Structures concepts, so one should be very careful as to how to avoid the creation of pointers that is no longer allocated. Dangling pointers are nasty bugs because they seldom crash the program until long after they have been created, which makes them hard to find. Programs that create dangling pointers often appear to work on small inputs, but are likely to fail on large or complex inputs.

```

delete [] s1;
delete [] s2;
return f(s1, s2); // s1 and s2 are dangling pointers

```

6.7.3 *Avoidance Of Memory Leak*

A memory leak is a particular kind of unnecessary memory consumption by a computer program, where the program fails to release memory that is no longer needed.

A memory leak can diminish the performance of the computer by reducing the amount of available memory. Memory allocation is normally a component of the operating system, so the result of a memory leak is usually an ever growing amount of memory being used by the system as a whole, not merely by the erroneous process or program. Eventually, in the worst case, too much of the available memory may become allocated and all or part of the system or device stops working correctly or the application fails.

6.7.4 *Unreachable Memory in C++*

Consider the following example and notice that memory was available and pointed to by `s`, but not saved. After this function returns, the pointer is destroyed and the allocated memory becomes unreachable to "fix" this code, you would add the statement `free(s)` to the else block before the "return 0" statement.

```
int f(void)
{
    char* s;
    s = malloc(50); /* get memory */
    if (s==NULL)
        return 1; /* no memory available */
    else
    { /* memory available */
        return 0; /* memory leak - see note below */
    }
}

int main(void)
{
    /* this is an infinite loop calling the above function */
    while (1)
```


(Build 2600.xpsp_sp2_rtm.040803-2158 : Service Pack 2)

Copyright © 1981-2001 Microsoft Corporation

• C++ Compiler : Turbo C++ Version 3.0

Copyright © 1991, 1995 Borland International, Inc.

CONCLUSION

Review

The softwares are generating correct results and the performance parameters are ok. They are also behaving properly and are not hanging the computer. The load distribution between the two cores of the dual core scheduler software is equal. The comparison of dual core scheduler with the single core scheduler software was a bit tedious because of simultaneous opening of two files and manual reading of data entries, but that has been taken care of by the graphical utility. All requirements that we had set out to satisfy have been met and we successfully developed a dual core operating system scheduler.

Future Work

This work can be extended by enlarging its scope and encompassing all known scheduling algorithms for the single core scheduler. Single core processors are in wide spread use around the world and the theory of their schedulers should continue to attract the attention of researchers and commercial establishments alike. In the dual core schedulers' domain a new algorithm, better than the one presented in this project, can be devised and its comparison can be done with the existing algorithms at that point in time.

BIBLIOGRAPHY

Galvin, Gagne, et al. *Operating Systems Concepts Windows XP Update*. John Wiley & Sons New York 2003

Dietel & Dietel. *C++ How to Program Third Edition*. Pearson Education Singapore 2001

www.microsoft.com

www.amd.com

www.intel.com

ALGORITHM.

Any well-defined sequence of steps (procedure or routine) that takes some value as input and guarantees a value as output in some finite number of steps.

ALLOCATE

To reserve a resource, such as sufficient memory, for use by a program.

ARCHITECTURE

1. The physical construction or design of a computer system and its components. See also cache, CISC, closed architecture, network architecture, open architecture, pipelining, RISC. 2. The data-handling capacity of a microprocessor. 3. The design of application software incorporating protocols and the means for expansion and interfacing with other programs.

BAR CHART

A type of graphic in which data items are shown as rectangular bars. The bars may be displayed either vertically or horizontally and may be distinguished from one another by color or by some type of shading or pattern. Positive and negative values may be shown in relation to a zero baseline. Two types of bar charts are common: a standard bar chart, in which each value is represented by a separate bar, and a stacked bar chart, in which several data points are "stacked" to produce a single bar. Also called bar graph.

C++

An object-oriented version of the C programming language, developed by Bjarne Stroustrup in the early 1980s at Bell Laboratories and adopted by a number of vendors, including Apple Computer and Sun Microsystems.

CENTRAL PROCESSING UNIT

The computational and control unit of a computer. The central processing unit is the device that interprets and executes instructions. Mainframes and early minicomputers contained circuit boards full of integrated circuits that implemented the central processing unit. Single-chip central processing units, called microprocessors, made possible personal computers and workstations. Examples of single-chip central processing units are the Motorola 68000, 68020, and 68030 chips and the Intel 8080, 8086, 80286, 80386, and i486 chips. The central processing unit--or microprocessor, in the case of a microcomputer--has the ability to fetch, decode, and execute instructions and to transfer information to and from other resources over the computer's main data-transfer path, the bus. By definition, the central processing unit is the chip that functions as the "brain" of a computer. In some instances, however, the term encompasses both the processor and the computer's memory or, even more broadly, the main computer console (as opposed to peripheral equipment). See also microprocessor. Acronym: CPU.

CLASS

In object-oriented programming, a generalized category that describes a group of more specific items, called objects, that can exist within it. A class is a descriptive tool used in a program to define a set of attributes or a set of services (actions available to other parts of the program) that characterize any member (object) of the class. Program classes are comparable in concept to the categories that people use to organize information about their world, such as animal, vegetable, and mineral, that define the types of entities they include and the ways those entities behave. The definition of classes in object-oriented programming is comparable to the definition of types in languages such as C and Pascal. See also object-oriented programming.

CLOCK RATE

The rate at which the clock in an electronic device, such as a computer, oscillates. The clock rate is normally given in hertz (Hz, one cycle per second), kilohertz (kHz, one thousand cycles per second), or megahertz (MHz, one million cycles per second).

Clock rates in personal computers increased from about 5 MHz to about 50 MHz between 1981 and 1995. Also called clock speed, hertz time.

CODE

1. Program instructions. Source code consists of human-readable statements written by a programmer in a programming language. Machine code consists of numerical instructions that the computer can recognize and execute and that were converted from source code. See also data, program. 2. A system of symbols used to convert information from one form to another. A code for converting information in order to conceal it is often called a cipher. 3. One of a set of symbols used to represent information.

COMPILER

A programming tool that translates a program written in a familiar high-level language like Basic, C++, or Java, into, typically, the machine language of a computer, which is composed only of zeroes and ones.

COMPUTER GRAPHICS

The display of "pictures," as opposed to only alphabetic and numeric characters, on a computer screen. Computer graphics encompasses different methods of generating, displaying, and storing information. Thus, computer graphics can refer to the creation of business charts and diagrams; the display of drawings, italic characters, and mouse pointers on the screen; or the way images are generated and displayed on the screen.

COMPUTER SCIENCE

The science concerned with the study of computational processes and with the design and implementation of hardware and of software to solve problems, characteristically by means of algorithms (or effective procedures) implemented in the form of programs.

CONCURRENT EXECUTION

The apparently simultaneous execution of two or more routines or programs. Concurrent execution can be accomplished on a single process or by using time-sharing techniques, such as dividing programs into different tasks or threads of execution, or by using multiple processors. Also called parallel execution. See also parallel algorithm, processor, sequential execution, task, thread (definition 1), time-sharing.

CONTEXT SWITCHING

A type of multitasking; the act of turning the central processor's "attention" from one task to another, rather than allocating increments of time to each task in turn. See also multitasking, time slice.

DATA STRUCTURE

A way to store and organize data in order to facilitate access and modifications.

DEBUGGING

The process of eliminating errors (or "bugs") from a computer program.

DEMO

1. Short for demonstration. A partial or limited version of a software package distributed free of charge for advertising purposes. Demos often consist of animated presentations that describe or demonstrate the program's features. See also crippled version. 2. A computer in a store that is available for customers to test, to see if they wish to buy it.

EXECUTABLE

A program file that can be run, such as file0.bat, file1.exe, or file2.com.

FILE

A physical unit of storage on a computer disk or tape.

GRAPHICAL USER INTERFACE

A type of environment that represents programs, files, and options by means of icons, menus, and dialog boxes on the screen. The user can select and activate these options by pointing and clicking with a mouse or, often, with the keyboard. A particular item (such as a scroll bar) works the same way to the user in all applications, because the graphical user interface provides standard software routines to handle these elements and report the user's actions (such as a mouse click on a particular icon or at a particular location in text, or a key press); applications call these routines with specific parameters rather than attempting to reproduce them from scratch. Acronym: GUI.

INITIALIZATION

The process of assigning initial values to variables and data structures in a program.

INPUT

Information entered into a computer or program for processing, as from a keyboard or from a file stored on a disk drive.

INTERUPPT

A request for attention from the processor. When the processor receives an interrupt, it suspends its current operations, saves the status of its work, and transfers control to a special routine known as an interrupt handler, which contains the instructions for dealing with the particular situation that caused the interrupt. Interrupts can be generated by various hardware devices to request service or report problems, or by the processor itself in response to program errors or requests for operating-system services. Interrupts are the processor's way of communicating with the other elements that make up a computer system. A hierarchy of interrupt priorities determines which interrupt request will be handled first if more than one request is made. A program

can temporarily disable some interrupts if it needs the full attention of the processor to complete a particular task. See also exception, external interrupt, hardware interrupt, internal interrupt, software interrupt.

L1 CACHE

A memory cache built into i486 and higher-level processors. The L1 cache, typically containing 8 KB, can be read in a single clock cycle, so it is tried first. The i486 contains one L1 cache; the Pentium contains two, one for code and one for data. Also called level 1 cache, on-chip cache. See also cache, i486DX, Pentium. Compare L2 cache.

L2 CACHE

A memory cache consisting of static RAM on a motherboard that uses an i486 or higher-level processor. The L2 cache, which typically contains 128 KB to 1 MB, is faster than the system DRAM but slower than the L1 cache built into the CPU chip. Also called level 2 cache. See also cache, dynamic RAM, i486DX, static RAM. Compare L1 cache.

LOGIC

The study of arguments, which are usually separated into the categories of deductive and inductive. The first system of logic was that of classical term logic, formalized by Aristotle, which studied the validity of arguments that can be formulated by means of a restricted class of sentences having specific kinds of logical form. Classical term logic characterizes the conclusions that follow from one premise (called immediate inference) and the conclusions that follow from two premises (called syllogistic inference), when premises and conclusions are restricted to so-called categorical sentences. Until around the mid-nineteenth century, Aristotelian logic was widely viewed as exhaustive of the subject. But the introduction of the sentential function by Gottlob Frege revolutionized the subject, and today Aristotelian logic is recognized to be only a special and relatively modest fragment of modern logic, which includes sentential logic (or the study of arguments when whole sentences are the basic units

of analysis) and predicate logic (or the study of arguments when sentences are analyzed on the basis of their internal structure). Although elementary logic is exclusively extensional (or "truth functional"), advanced logic pursues the formalization of intensional relations that are not merely truth-functional, including the nature of subjunctive, causal, and probabilistic conditionals, but also set theory, recursive function theory, and the theory of models.

OBJECT-ORIENTED PROGRAMMING

A currently popular programming paradigm, based on the principles of data abstraction, that de-emphasizes traditional algorithmic forms of program control in favor of the notions of classes, objects, and methods.

OPERATING SYSTEM.

The special software required to make a computer work. It provides the link between the user and the hardware. Popular operating systems include: DOS, MacOS, VMS, VM, MVS, UNIX, and OS/2. (Note that "Windows 3.x" is not an operating system as such, since it must have DOS to work.)

PARALLEL COMPUTING

The use of multiple computers or processors to solve a problem or perform a function. See also array processor, massively parallel processing, pipeline processing, SMP.

PARAMETER

A variable, belonging to a subroutine, which receives a value when the subroutine is executed.

PROGRAM

A set of instructions that controls the operation of a computer. The concept of a program is highly ambiguous, since the term "program" may be used to refer to (i) algorithms, (ii) encodings of algorithms, (iii) encodings of algorithms that can be

compiled, or (iv) encodings of algorithms that can be compiled and executed by a machine. As an effective decision procedure, an algorithm is more abstract than a program, since the same algorithm might be implemented in various specific programs suitable for execution by various specific machines by using various programming languages. From this perspective, the senses of "program" defined by (ii), (iii), and (iv) provide conceptual benefits that definition (i) does not.

REAL TIME OPERATING SYSTEM

An operating system designed or optimized for the needs of a process-critical environment.

SCHEDULER

An operating-system process that starts and ends tasks (programs), manages concurrently running processes, and allocates system resources. Also called dispatcher.

UNIX

A multiuser, multitasking operating system originally developed by Ken Thompson and Dennis Ritchie at AT&T Bell Laboratories in 1969 for use on minicomputers. UNIX is considered a powerful operating system that, because it is written in the C language, is more portable--that is, less machine-specific--than other operating systems. UNIX is available in several related forms, including AIX (a version of UNIX adapted by IBM to run on RISC-based workstations), A/UX (a graphical version for the Apple Macintosh), and Mach (a rewritten but essentially UNIX-compatible operating system for the NeXT computer). See also BSD UNIX, GNU, Linux.

UTILITY PROGRAM

A program designed to perform maintenance work on the system or on system components (e.g., a storage backup program, disk and file recovery program, or resource editor).

VARIABLE

An entity in a computer program whose role is to hold arbitrary data.