NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AN APPROACH TO INTEGRATION OF REAL-TIME
SOFTWARE FOR AN AUTONOMOUS UNDERWATER
VEHICLE

by

Brent Lee Leatherman
June 1991

Thesis Advisor: Shridhar Shukla

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	

4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b. OFFICE SYMBOL (If applicable) 32	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000		7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000

8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS			
	Program Element No	Project No	Task No	Work Unit Accession Number

11. TITLE (Include Security Classification)
AN APPROACH TO INTEGRATION OF REAL-TIME SOFTWARE FOR AN AUTONOMOUS UNDERWATER VEHICLE

12. PERSONAL AUTHOR(S) Leatherman, Brent, L.

13a. TYPE OF REPORT Master's Thesis	13b. TIME COVERED From To	14 DATE OF REPORT (year, month, day) June 1991	15. PAGE COUNT 76

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES			18 SUBJECT TERMS (continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUBGROUP	Real-time, multi-tasking, Rate Monotonic Scheduling, AUV

19. ABSTRACT (continue on reverse if necessary and identify by block number)

The Naval Postgraduate School (NPS) is currently involved in a long term project to investigate and develop real-time software for command and control of Autonomous Underwater Vehicles (AUV). In support of this goal, NPS is currently designing and fabricating a testbed AUV. This thesis describes the design development, and testing of a real-time scheduling software package to act as the top layer of control software for the AUV. Also discussed are the various real-time scheduling policies available along with the features of the assigned operating system that allow the implementation of the scheduler.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS REPORT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a NAME OF RESPONSIBLE INDIVIDUAL Shridhar Shukla	22b. TELEPHONE (Include Area code) (408) 647-2057	22c OFFICE SYMBOL EC/Sh

DD FORM 1473, 84 MAR   83 APR edition may be used until exhausted   <u>SECURITY CLASSIFICATION OF THIS PAGE</u>
All other editions are obsolete   Unclassified

AN APPROACH TO INTEGRATION OF REAL-TIME SOFTWARE FOR AN
AUTONOMOUS UNDERWATER VEHICLE

by

Brent Lee Leatherman
Lieutenant, United States Navy
B.S., University of Florida

Submitted in partial fulfillment of the
required of degree for

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

The Naval Postgraduate School (NPS) is currently involved in a long term project to investigate and develop real-time software for command and control of Autonomous Underwater Vehicles (AUV). In support of this goal, NPS is currently designing and fabricating a testbed AUV.

This thesis describes the design, development and testing of a real-time scheduling software package to act as the top layer of control software for the AUV.

Also discussed are the various real-time scheduling policies available along with the features of the assigned operating system that allow implementation of the scheduler.

I. INTRODUCTION

A. GENERAL

The Naval Postgraduate School (NPS) is developing an autonomous underwater vehicle (AUV) - an untethered, intelligent, robot submarine. The current AUV, AUV-II, is the second in a series of three vehicles to be built. It has been designed as a testbed for research in adaptive control, mission planning and execution, data collection and analysis, and the uses of various sensors.

AUV-II is driven by a pair of 4" propellers and four hovering thrusters. These and the eight control surfaces used allow for a high degree of vehicle control. The vehicle's sensors include four sonars, a depth cell, and a speed indicator. Positional data is provided by three on-board gyroscopes and a magnetic compass.

The on-board computer hardware is centered around a twelve slot G-96 bus supplied by the GESPAC corporation [DIBBLE 90]. The bus hosts a Motorola 68030-based processor board along with 2.5 Megabits of RAM and 4 Megabytes of EPROM. Five other slots are dedicated to a 200 Megabyte hard disk drive, parallel and serial communication ports, an analog-to-digital channel from the sensors, and a digital-to-analog channel to the effectors. The remaining slots are expected to

be used for additional memory and Transputer boards. Figure
1.1 [HEALY 90] shows the internal and external configurations
of AUV-II.
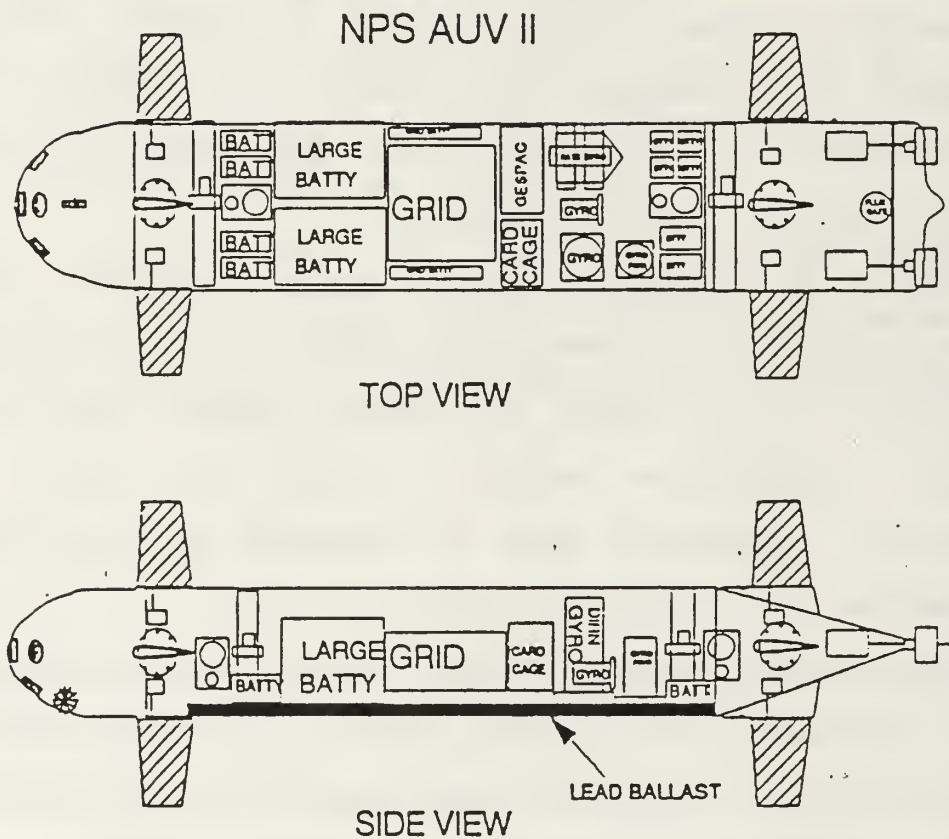
Figure 1.1 Internal and External Configurations of AUV-II

The computer uses the OS-9 operating system developed by MICROWARE for GESPAC [GESPAC 88]. OS-9 is a complete operating system having a file system, native compilers, a built-in editor, the ability to support local area networks, multi-tasking abilities, real-time features such as timers, process creation and deletion primitives, process priority assignments, and signal sending and receiving structures. It has the added advantage of having been used in prior AUV research [HEALY 90]. The AUV-II project is ongoing, and has had many successful trials in the NPS pool.

B. COMPUTATIONAL REQUIREMENTS OF THE AUV

For successful mission completion, AUV-II must be able to execute a complex set of both periodic and aperiodic processes (or tasks) that may have widely varying attributes such as variable execution times, frequencies and response times. These tasks may include: mission planning, path planning, guidance, effector control, sensor control, collision avoidance, and data collection [HEALY 90]. The inter-relationship of these are shown in Figure 1.2 [HEALY 90]. As can be seen in Fig. 1.2, the operator starts, and controls the entire mission. The focus of this thesis is to provide real-time control for that mission, consisting of many tasks with widely differing computational requirements. For example, block 8, collision avoidance, is an aperiodic task - since obstacles cannot be predicted - while block 6, navigation,

predicted - while block 6, navigation, requires periodic computer attention. Also, while both block 6 and block 5, sonar, are periodic, their respective periods are quite different; sonar having the higher frequency. Since obstacle avoidance has not yet been completely integrated in the AUV, this thesis concentrates on periodic task scheduling and execution. Once the parameters of aperiodic tasks have been finalized, it should require very little extra work to incorporate aperiodic scheduling into the existing control software. A more detailed analysis of Figure 1.2 may be found in [HEALY 90]. Computational resources must be used such that all tasks complete execution within their respective periods, or, in real-time terms, meet their deadlines in a predictable and reliable manner regardless of the difference in their attributes.

In the current vehicle software, all processes are executed in an infinite loop operating at roughly 10 Hz. Although this is adequate for the simple missions now being executed, it lacks the necessary sophistication for the more complex missions planned. These missions may include, for instance, tasks that are required to be executed at different frequencies. For example the sonar system operates at 5 Hz while the motion control loop operates at 10 Hz.

A recent study [BIHARI 90] of the processing requirements of this project delineates the necessity of an efficient, hard real-time scheduler to ensure the correct functioning of the

AUV DATAFLOW DIAGRAM
Baseline System
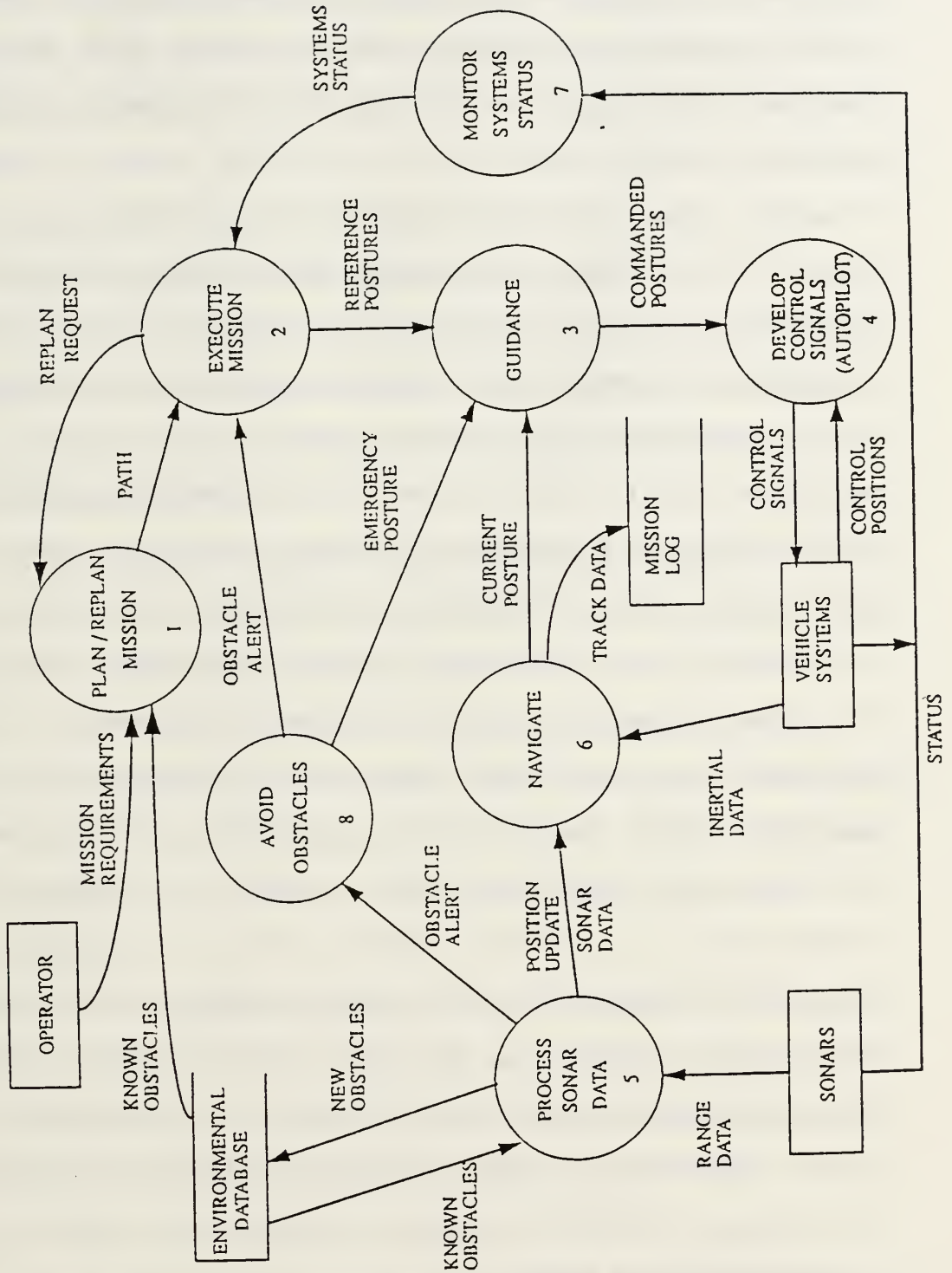
Figure 1.2 Data Flow Diagram for the AUV

vehicle. A scheduler is the top layer of AUV-II software, which coordinates the task set to ensure that all process deadlines are met in a reliable and predictable manner. The scheduler must be able to, at a minimum, access at least one accurate time base, run processes at specified points in time, support synchronous and asynchronous communications between processes, and place bounds on the response time of aperiodic processes and events. An important attribute of a scheduler is how it responds to an overload condition in which it is not possible for all tasks to meet their deadlines because they exceed the net processing reserves available. Overload may occur in two ways, **transient** or **sustained**. Transient overload is caused by the execution of an aperiodic event that requires the suspension of one or more periodic events. Sustained overload is caused by unexpected increases in process execution time such that the processor load exceeds its resources. A scheduler must be able to handle overload conditions in a stable manner. For a scheduler to be considered stable, two requirements must be met. The first being that a subset of the most critical tasks aways meet their respective deadlines. The second is a quick return to normal operation once the overload condition has passed.

C. OBJECTIVES OF STUDY

OS-9 scheduling policy cannot provide hard real-time performance without the addition of a control layer of

6

software to schedule the task sets required by the AUV. Therefore, the main objective of this study is to develop a real-time scheduler that meets the following requirements:

1. It must use only OS-9 real-time primitives, with no modifications to the operating system.

2. It must be capable of maintaining a relatively high processor utilization without missing task deadlines.

3. It must be able to support task sets with varying characteristics.

4. It must remain stable under transient overload. Rapid recovery after elimination of the overload is of prime importance in this case.

5. It must remain stable and degrade gracefully under sustained overload. In this case, the important feature is meeting the deadlines of a subset of more critical processes while sacrificing those that are less critical.

6. Since the scheduler will be implemented on an already functioning vehicle, it must be able to integrate and communicate with the existing motion control software with as few modifications as possible to that software. AUV software modules are being designed by different project members that have little knowledge of the scheduler; therefore, it must be simple to use and maintain. In order to simplify the use of the scheduler,

a graphical interface is being developed and is the
subject of a follow-on thesis.

D. METHOD OF APPROACH

This thesis deals with the implementation of a theoretical
method of real-time scheduling into a practical working
scheduler robust enough to handle the various situations that
are to be expected on a functioning vehicle. As such, not only
must the scheduler meet the requirements of the selected
scheduling policy, but also account for the real world
conditions such as process switching overhead and variable
process execution times that will degrade performance below
theoretical values. It was necessary to determine what
processor loads were feasible and what characterization of
processor set provided the required performance. To facilitate
this implementation the following steps were taken:

1. Selection of a suitable scheduling policy.
2. Implementation of the policy using OS-9 dependent
   resources.
3. Experimental determination of the available utilization
   and observation of scheduler stability under various
   loads and task set characteristics.

All programming for this project was done using the resident C compiler provided by MICROWARE, and is closely tied to the OS-9 operating system [MICROWARE 87]. Porting to another system would require extensive modification of the scheduler.

This project successfully implemented a scheduling layer of software that can determine if a given periodic task set can be successfully scheduled to deliver processor utilization exceeding 85%. Aperiodic tasks can be scheduled on a more limited basis by treating them as periodic events with a period equal to infinity. The improvement over the passive OS-9 policy is clearly demonstrated in a later chapter of this thesis.

E. THESIS ORGANIZATION

Chapter II presents a discussion of various real-time scheduling policies and outlines the reasons for using rate monotonic scheduling for AUV-II. Chapter III is a more complete and detailed study of rate monotonic scheduling theory. Chapter IV details the implementation of rate monotonic scheduling using the OS-9 operating system. Chapter V presents the methods and experiments used to determine scheduler performance and veracity. Chapter VI summarizes the results of this study and describes future directions for study.

II. AN OVERVIEW OF REAL-TIME SYSTEMS

A. GENERAL

A real-time system is defined to be one in which the various processes adhere to time constraints determined by the application. Although real-time systems are often associated with extremely fast processing speeds, this is not an accurate interpretation. A real-time system refers to the timely occurrence of events and the intervals involved may range from microseconds to years [LEVI 90].

Real-time systems can be loosely divided into two categories, **hard deadline** and **soft deadline** systems. Hard deadline systems are those where the missing of a process deadline is catastrophic. Conversely, while making all deadlines is desirable on a soft deadline system, missing one is not a fatal error [LEVI 90].

Assigning priorities to different concurrent processes is a common mechanism used to share computational resources in a real-time system. Based on how the priorities may be assigned, real-time systems can be further classified as fixed priority systems in which all process priorities are fixed at run time, or dynamic priority systems in which priorities can be changed during execution. Rate monotonic scheduling [LIU 73] is an example of a scheduling policy that uses fixed prioritization.

10

Conversely, earliest deadline scheduling policy [LEVI 90] makes use of dynamic scheduling for successful operation. If these two types of systems are combined, a hybrid is achieved that is known as a mixed priority system [LEVI 90].

B. OS-9 SPECIFIC REAL-TIME PRIMITIVES

The OS-9 operating system contains a wide range of real-time primitives that can be used to implement a functioning real-time scheduler. This section describes these primitives in a general fashion; a detailed listing of the commands and their functions may be found in Appendix D.

OS-9 is, inherently, a multi-tasking system. This is demonstrated by the process priority assignment embedded in OS-9. Processes may be assigned a priority from one of 65,536 separate priority levels when they are forked into existence. The forked processes are placed on a ready-to-execute queue in direct order of their priority. The process at the head of the queue is executed for one or more ticks. A tick is the smallest unit of time the system recognizes and may be set by the system operator for a duration as low as 10 milliseconds. Once a process has started executing, it will execute until it has completed execution or until the end of the current tick. All non-executing processes have their priorities increased by one for each tick spent in the ready-to-execute queue, a process referred to as aging [MICROWARE 87]. Aging is normally

11

used to ensure that all processes eventually receive the attention of the processor regardless of their priority.Once a process has executed, it is reduced to its original priority. The queue is checked every tick to determine which process has the highest priority and, therefore, the right to the processing unit. Process switching times are on the order of 50 microseconds [DIBBLE 88].

OS-9 uses this queue and various commands to support real-time operations. These commands include timing commands that check absolute, relative, and elapsed times. These commands all have a resolution of a tick. System commands are also provided that allow processes to suspend themselves. A suspended process is removed from the ready-to-execute queue and does not age until reactivated. Reactivation can be accomplished by either an elapsed time counter, an absolute time event or upon the receipt of a signal from another process. Signal sending and receiving mechanisms are closely coupled with process suspension and are used to provide interprocess coordination and timing. Signals can be used to either wake up or kill a process. Customized signals are also supported. Another class of primitives that help in the development of real-time systems are the ones that allow for access to process identities, states, and parameters [GESPAC 88].

C. IMPLEMENTATION

The rate monotonic scheduling algorithm was chosen as the method to implement upon AUV-II, due to it being the optimal fixed priority scheduling algorithm [LIU 73]. While dynamic scheduling can increase overall processor utilization, it is much more difficult to implement in a practical fashion due to the problem most operating systems have in arbitrarily reassigning priorities during execution [LIU 73]. Rate monotonic scheduling is presented in greater detail in the following chapter.

III. RATE MONOTONIC SCHEDULING THEORY

A. INTRODUCTION

The rate monotonic scheduling (RMS) algorithm was chosen as the basis for the scheduler on AUV-II; its practical aspects having been recently analyzed [SHA 90]. RMS uses the following model of the workload.

1. The workload must consist of independent processes; an independent process being one that does not require the execution of any other process to function.

2. All process priorities are fixed at the time of execution.

3. The system must provide for pre-emptive execution; i.e. higher priority processes will interrupt the execution of lower priority processes.

Given these conditions, RMS guarantees that in the worst case conditions, all deadlines will be met for at least a 69.3% processor load. Processor load is defined to be the sum of the execution times of each process divided by the respective period of each process. This function is shown in Equation (3.1).

$$X = \frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_n}{T_n}$$

where

X is the processor load
C_i is the execution time of process i
T_i is the period of process i
n is the number of processes

(3.1)

The above mentioned limit is known as the Liu-Layland bound named for the two men who derived it [LIU 73].

B. PERIODIC TASK SCHEDULING

One of the most important attributes of a task set is its phasing; the order and timing in which the various tasks are to be executed. The Liu-Layland bound represents the worst case task set phasing; i.e. one with the maximum of number of task preemptions. A system may be able to support a higher processor utilization if the task set phasing is more favorable. Two theorems have been developed to determine if the higher processor load is schedulable [LEHOCZKY 89]. For a task set to be considered schedulable, all tasks must meet their respective deadlines under all task set phasings.

The first method used in determining schedulability is given by Equation (3.2) [SHA 90].

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1)$$

where   (3.2)

C_i is the execution time of process i
T_i is the period of process i
n is the number of processes

The upper limit of 1.0 decreases monotonically until, as n approaches infinity, the right side of Equation (3.2) yields the Liu-Layland bound of 0.693, or, 69.3%. If the processor load is less than or equal to this bound, the task set is always schedulable. The second method for determining schedulability is more complex, but more exact, and is given in Equation (3.3). The term **scheduling points** in Equation (3.3) refer to the arrival times for all processes in the task set within the longest period in the task set. For example, a task set consisting of four tasks periods 4, 6, 12, and 25 units, would have the following set of scheduling points: 4, 6, 8, 12, 16, 18, 20, 24, and 25 [SHA 86].

$$\forall i, 1 \leq i \leq n$$

$$\sum_{j=1}^{1} C_j \frac{1}{lT_k} \lceil \frac{lT_k}{T_j} \rceil \leq 1.0$$

$$(k, l) \, \varepsilon R_i$$

where   (3.3)

C_j is the execution time of process j
T_j is the period of process j
R_i is the collection of scheduling points for the task set
l is the ratio of $\lfloor \frac{T_i}{T_j} \rfloor$

16

Equation (3.3) is evaluated for the entire set of scheduling points and the results compared to the value 1.0. If any of these values are less than, or equal to, 1.0, the task set is always schedulable under RMS. Once schedulability has been determined, RMS assigns a fixed priority to each process. This assignment is in inverse proportion to the period of the process; the shorter the period, the higher the priority. Once priorities have been assigned, the processes can be executed using the preemption functions provided by the system. Although neither of the two methods of determining schedulability guarantees more than a 69.3% utilization under worst case conditions, average case utilization values can range from 85-90%, well above the Liu-Layland bound. This higher utilization value is attributable to the concept of the **critical instant** [SHA 90]. A critical instant is one in which all processes are initiated at the same instant, thus creating the maximum number of preempted processes. It has been determined that the higher the number of critical instants, the lower the achievable utilization [SHA 86]. This then can be used to predict worst case performance and, by judicious selection of task periods (assuming some flexibility in period selection), a task set can be 'tuned' to achieve high utilization.

C. APERIODIC TASK SCHEDULING

Aperiodic tasks are tasks that appear on an irregular,

unpredictable basis. An example of an aperiodic process for the AUV-II would be collision avoidance. RMS services aperiodic tasks using a sporadic server; that is, a mechanism that executes the task if a request is sent from the aperiodic task, and if there are sufficient reserve computational resources to process the request. In other words, the system workload must be low enough that the aperiodic event can be served without disrupting the periodic task set. Provisions must be made for the case where the aperiodic event is crucial and therefore needs to be able to preempt all, or part, of the periodic task set [SHA 90].

C. PERIOD TRANSFORMATION

RMS always guarantees that a certain subset of the task set will meet their deadlines under worst case conditions. This subset is known as the critical set. Since process priority is in inverse proportion to period lengths, under some circumstances, a process with a long period and lower priority may serve a more critical function than a process with a shorter period and must be included in the critical subset [SHA 90].

This problem is solved using a technique known as **period transformation**. This method requires splitting the critical process into two logical parts, each one having 1/2 the worst case execution time of the process and 1/2 the requested period. Priority is then assigned to the process according to

18

the reduced period length and, thus, elevated. Once task set
execution begins, the critical process executes 1/2 of its
worst case execution time in the first half of its period and
then suspends itself. The other half of the execution comes
during the second half of its main period. This is shown in
Figure 3.1 [SHA 90].

```
0                      T/2                              T
|                       |                               |
|   +---------------+   |    +---------------+          |
|   |               |   |    |               |          |
|   |  FIRST PART   |   |    |  SECOND PART  |          |
|   |               |   |    |               |          |
|   |    OF P1      |   |    |    OF P1      |          |
|   |               |   |    |               |          |
|   +---------------+   |    +---------------+          |
```
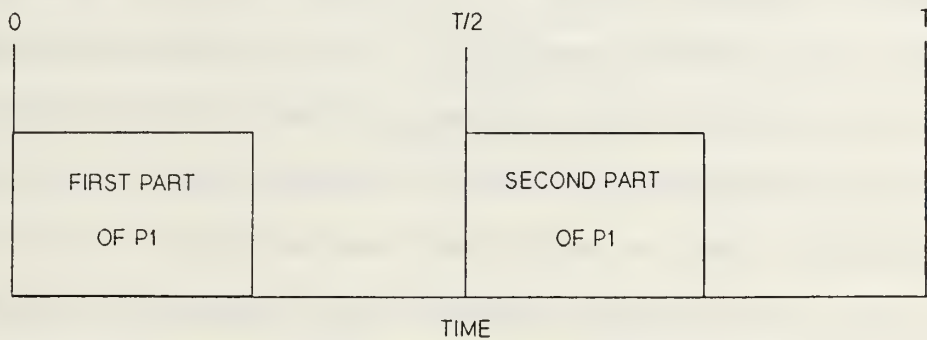
TIME

Figure 3.1 Time Line Showing Period Transformation

Given the limitations of the OS-9 operating system, some of
the features of RMS were more difficult to implement than
others. The RMS attributes implemented in this thesis include
schedulabilty checking, periodic process serving, a limited
ability to service aperiodic tasks, and the ability to remain
stable under overload conditions. The details of this
implementation are described in the next chapter.

IV. AN IMPLEMENTATION OF RMS ON OS-9

A. GENERAL

The scheduling software, as implemented on the GESPAC computer, consists of five logical parts. These are: the scheduler startup process, initialization, schedulability analysis, the master scheduler, and the modifications that need to be made to the processes called by the scheduler. Due to the limitations of the OS-9 operating system, and the specification that no modifications be made to this system, this was a non-trivial problem. As mentioned before, the scheduler is highly OS-9 dependent; therefore, there has been an effort made to highlight all non-standard C commands used in order that future modifications to the scheduler can be accomplished without excessive difficulty. Figure 4.1 illustrates the sequence of steps in the scheduling software. These are more fully explained in sections B through F of this chapter.

B. SCHEDULER STARTUP

To function correctly, the run time module must have the highest priority on the system. This is due to its function as the coordinator of the AUV-II software; in other words, it must not be preempted by a called process while it is
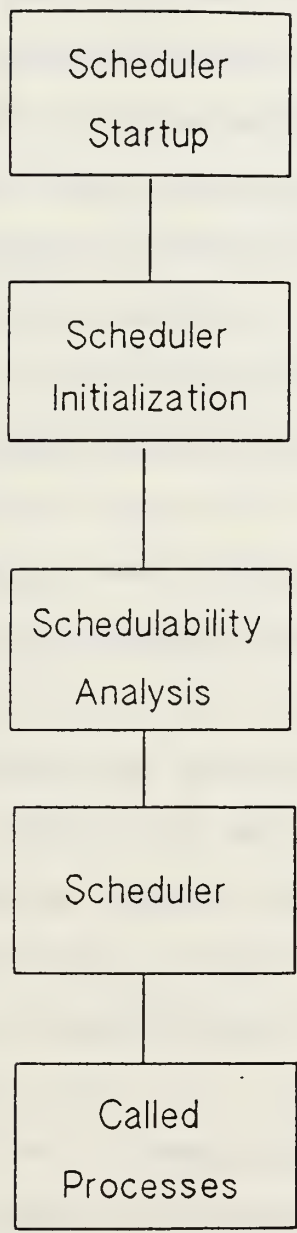
Figure 4.1 Step Sequence of Scheduling Software

executing the scheduling process. In the OS-9 operating system, run-time priority is set by the command, **os9forkc**, and cannot be changed in the process itself. Thus, it was necessary to develop a small calling program to establish the scheduler's initial high priority. It first spawns the scheduler and then suspends itself until the completion of the mission. It should be stated that all processes to be called using the os9forkc command are required to have an executable module located in a directory available to the scheduler. Since they will be called by OS-9 commands, the called processes should not be linked together during the compilation process.

As stated in Chapter II, OS-9 provides for 65,536 distinct priority levels. It was arbitrarily determined that a priority level for the scheduler of 62,000 would be appropriate. This is high enough to preempt all other system processes, but still leaves some higher priorities for emergency processes such that the scheduler would be deliberately interrupted. This priority can be easily changed in the startup module. The source code for the startup module is located in Appendix A.

C. SCHEDULER INITIALIZATION

Scheduler initialization begins by entering task set parameters. The first parameter is the number of processes in the task set. Once this has been entered at the keyboard, a loop is started that brings in the names of the processes,

their periods and execution times. Although this is currently entered manually at the keyboard, it may be changed to a file input/output system for implementation on the vehicle if desired. Process execution times can be obtained in two ways; the first is direct input by the system operator, the second is done by the scheduler as it forks slightly modified versions of the modules and times their lengths of execution. This procedure is outlined in Appendix B.

Once the task set parameters are available to the scheduler, the program sorts the processes by period from shortest to longest with ties broken arbitrarily. This is accomplished by a standard heap sort routine. This sort also establishes the relative priorities of the process from highest to lowest in inverse proportion to the length of the respective periods.

Absolute priorities must be assigned to each process before they can be forked off for execution. Due to the inherent conflict between the fixed priority requirements of RMS and the automatic aging of non-executing processes as implemented by OS-9, it became necessary to space process priorities far enough apart to ensure that no processe priority could overtake another. This scheme successfully negates the run-time aging of process priorities inherent to OS-9 and gives the illusion of fixed priorities to the scheduler.

Absolute priority assignment is accomplished by first

finding the least common multiple of the periods of all processes in the task set. Once the least common multiple is established, priorities are set using Equation (4.1).

$$P_i = P_0 - \left\lceil \frac{\left(\prod_{j=1}^{i} C_j\right)}{\left(\prod_{j=1}^{i} T_j\right)} \; LCM \right\rceil$$

where (4.1)

P_i is the processes priority
P_0 is the highest priority process
C_j is the execution time of process j
T_j is the period of process j
LCM is the least common multiple of all periods

The process with the shortest period, and therefore the highest priority, is assigned a priority of 60,000. This is P and was chosen to be well below the priority of the scheduler, while being large enough to permit a wide range of priorities under it. To illustrate the use of Equation (4.1), consider a task set consisting of two processes. Process one has a period of 50 ticks, and an execution time of 12 ticks. Process two has a period of 100 ticks, and an execution time of 15 ticks. Process one is automatically assigned a priority of 60,000, the maximum available. Process two has its priority assigned by subtracting the product of both execution times, a value of 180, and dividing it by the product of the periods, a value of 5000. This quotient is 0.036 and is multiplied by the LCM of the two periods, 100, giving a final value of 3.6. This is

24

rounded up to the next highest integer, 4.0, and subtracted from the value of the highest priority process. Process two, then, is assigned a priority of 59,996.

Although Equation (4.1) gives the correct absolute spacing between process priorities, under certain conditions, it can cause abnormally wide spacings between the priorities. This is usually caused by a large number of processes with different periods resulting in a large least common multiple. An example of this is given by considering two, two-process sets. Set one has two tasks with periods of 50 and 100 while set two has two task with periods of 51 and 100. Although these two sets are virtually identical, the first has a least common multiple of 100 while the second is 5100. When these least common multiples are used in Equation (4.1), the spacings would place the priorities of the second set much farther apart than in the first set. If there were an infinite supply of priority levels, this would be no problem, but, as mentioned earlier, OS-9 supplies only 65,536 priorities which are quickly spanned by a reasonably large task set. The solution to this problem was to first detect the large spacings and then reassign them the priority equal to the priority of next highest priority process minus the value equal to the number of ticks of the longest period in the task set. This value is theoretically the most any one process could age before it is executed. This two pronged approach to prioritization has proven to be successful in all tests and experiments using the scheduler.

There were no recorded cases of a lower priority process overtaking a higher priority process.

D. SCHEDULABILITY ANALYSIS

The schedulability of a task set uses the process described by theorem three [SHA 90]. The steps are described in a series of Equations (4.2-4.5) shown below.

The first step, given in Equation (4.2), is to determine the processor demands made by the task set as a function of time [Lehoczky 89]. The second step is to normalize the results to achieve a value between 0.0 and 1.0. This

$$W_i(t) = \sum_{j=1}^{i} C_j \lceil \frac{t}{T_j} \rceil$$

where

$W_i(t)$ is processor load as a function of time (4.2)
$1 \leq i \leq n$
n is the number of processes ϵ the task set
C_j is the execution time of process j
T_j is the period of process j
t is time as a continous variable

is accomplished in Equation (4.3) by dividing by time and is simply an intermediate step in the process of converting the results from continuous to discrete values. The next step is

$$L_i(t) = W_i(t)/t \qquad (4.3)$$

given is Equation (4.4) where the continuous time variable, **t**,

26

is converted, by the computation of scheduling points, to the discrete variable, **s**, as discussed in Chapter II.

$$L_i = \min(0 \le s_i \le T_i) L_i(s_i)$$

where   (4.4)

s_i *are the various scheduling points*

The final step is a check to determine if Li has a value less than or equal to 1.0, as shown in equation (4.5).

$$L_i \le 1.0$$ (4.5)

If the task set meets schedulability requirements, a confirmation message is sent to the system operator and the master scheduler initiated. If the task set does not meet schedulability requirements, an error message is sent to the operator and the scheduler is suspended until a workable task set is developed.

E. THE SCHEDULER

Once priorities have been calculated and schedulability determined, scheduling analysis is complete and task set execution may begin. The analysis and execution functions are currently part of the same software module. This means that analysis is done in the field prior to the beginning of the

mission. Since the task set has presumably been checked before the various field tests, only the executable module needs to be ported to the AUV, with the various values generated by the analyzer being read from a disk file prior to the start of a mission. This will require some minor modifications to the scheduler.

The first step is the creation of the individual processes that are listed in the task set. This is done using the OS-9 commands **os9forkc** and **os9exec**. The original program specifications had called for a scheme that created a process at the start of each period, executed the process and then killed the process prior to the end of the period. This proved to be unworkable when it was found that the procedure for creating a process took a random time varying between 220 and 340 milliseconds. Since at least one vehicle control module was required to run at 10 Hz, this method was fatally flawed. A more sophisticated approach was developed that entailed creating each process once, at the beginning of the scheduler module, and then, after execution, suspending the process until receipt of a signal from the master scheduler. This method reduced the task switching time to 55 microseconds, some 400,000% faster than the original method. The second method had the added benefit of reducing the amount of processor time conducting the 'garbage collection' required in the creation and killing of processes.

As the scheduler creates each process, two entries are

made in the table that is the heart of the scheduler and is
shown in Figure 4.2.

PROCESS ID NUMBER	TIME UNTIL NEXT EXECUTION
1	50
2	100
3	150
4	250
.	.
n	Tn

Figure 4.2 Table Data Structure with Sample Entries

The first column in the table is the identification number
of each process. The second column is the next time to
execution for each process. The master scheduler enters an
infinite loop, once all processes have been created and placed
on the ready-to-execute queue. Once in the loop, the scheduler
scans the second column for zero values which indicates that
the process is ready to start execution. If a zero entry is
found, the process identification number is recorded and the
scan continued through the table. Once all entries in the
table have been checked, a time stamp is taken and the flagged
processes are sent wake up signals from the scheduler. When
the last signal has been sent, another time stamp is taken and
the elapsed time from the start of the signaling procedure to
the end is recorded and all signaled processes have their next
time to execute entry reset to their original period. All

table entries then have the elapsed time subtracted from the second column of the table. This is done to prevent scheduler overhead from affecting the scheduling process. The scheduler then scans the second column for the smallest value present. This value is subtracted from all second column entries and the main scheduler is then suspended for this length of time to allow the called processes to begin execution. The possible conflicts between the children processes are resolved through the preemption allowed by the priorities assigned. This entire process, in psuedo-code, is illustrated in Figure 4.3.

```
While(true)
    scan table for zero entries;
    if entry = 0,
        record id number;
    take time stamp #1;
    for i=0 to m    /* m = number of processes to be
                            signaled */
    find process(i) id number;
    signal process(id number);
    end for loop
    take time stamp #2;
    adjustment = time stamp #2 -time stamp #1;
    for i = 0 to n /* all processes */
      table entry(i) = table entry(i) - adjustment;
    scan table for smallest entry (in ticks);
    for i=0 to n /*all processes */
     table entry(i) = table entry(i) - smallest entry;
    suspend scheduler for smallest entry ticks;
end while;
```

Figure 4.3 Psuedo Code for Signaling Process

During program development, the control loop was modified from an infinite loop to a finite loop controlled by the number of signals sent. It is recommended that onboard the AUV it remain an infinite loop with logic implemented to halt the loop from an outside source such as a keyboard interrupt or a time signal from the system clock or an elapsed time counter. This will allow for a more graceful exit from the program than the currently required rebooting of the system. A copy of the source code for the scheduler is found in Appendix B.

F. MODIFICATIONS OF CALLED PROCESSES

One goal in the original specifications for the scheduling software was that the other software modules would not have to be modified, that they could run 'as is.' This led to the original plan of create/execute/kill discussed earlier. When this proved unworkable, it was determined that a minimum amount of modification would have to be made in the other software modules in the system. These were designed to be quickly and easily installed in only a few minutes per module.

To implement the first modification, it is necessary to determine the sections of software that are to be executed each period versus those that are to be only run once, such as process initialization. Once these sections are identified an infinite while loop is installed around them. The second modification is to install the signal catching commands in the loop between the last local line of code and the end of the

31

infinite while loop. These modifications are demonstrated in
Figure 4.4. The command most suitable for this is the OS-9

```
main()
{
    initialization commands;
             .
             .
    while(1)
    {
        repeated commands;
             .
             .
        wait();
    }
    exit();
}
```

Figure 4.4 Modifications to Called Processes

command wait() with no parameters. If it becomes desirable to
return data or instructions back to the parent process, the
OS-9 command exit() could be used. The wait() command suspends
the child process (removes it from the ready-to-execute queue)
until it receives a signal from the scheduler. During normal
execution only two signals would be sent: a '1.0' to wake the
process up, or a '0.0' indicating that the process is to be
killed and its allotted memory returned to the system. The
'0.0' is normally sent at the end of the mission to ensure a
graceful termination of the scheduler.

The emphasis on the graceful conclusion of the scheduler
is based on practical experience on the OS-9 system. When a
process finishes without an exit() command, the process
identification table and the processes allotted memory remain

allocated by the system until the system is rebooted. It takes surprisingly few of these to fill and fragment the system memory. When the system is in this condition it will continue to attempt to function normally, but no processes will execute. This is annoying in a PC-based machine, and possibly dangerous to an operating vehicle.

The last modification is used if the operator intends to allow the scheduler to measure the execution times of the processes. Since the time only needs to be measured over one execution of the process the infinite loop is removed and the wait() command is replaced with the exit() command. The modified process is named the same as the original process with the letter 'a' appended to it. As with the original process a executable module must be available to the system.

G. APERIODIC EVENTS AND PERIOD TRANSFORMATION

Aperiodic events are handled by the scheduler in the current implementation by treating them as a periodic event with an infinite period. This method assumes that aperiodic events can be sent a wake up signal by a process other than the scheduler, whereupon it executes in the same fashion as a periodic event. Upon completion of the aperiodic event, the process suspends itself until signaled the next time. A more sophisticated approach would require modifications to the timing table in the master scheduler and the prioritization of the processes. This would entail marking the aperiodic events

with a flag and then schedule them in the same manner as periodic events with the proper priorities. The next step is done while scanning the table for executable processes and would require the addition of logic to check for the flag on aperiodic events and only signal them if circumstances warranted.

Period transformation was not implemented in the current scheduler due to limitations in the OS-9 operating system. To function correctly with a minimum of modification to the called processes, a method is needed to not only wake up the called processes from the master scheduler, but also to suspend the called processes. This is not possible; although a method has been derived in which period transformation could be implemented on, at least, a limited basis. This method would require that extensive modifications be made to the called processes and is beyond the scope of this study [SHUKLA 91].

Upon the completion of the scheduler software development, numerous tests and experiments were conducted to confirm the scheduler's veracity, reliability and limitations. These results are reported in the next chapter of this thesis.

V. EXPERIMENTS AND RESULTS

A. GENERAL

Once the scheduler was written, it was necessary to devise methods to ensure the scheduler's veracity and to demonstrate the improvement it offered over passive OS-9 as installed. It was also necessary to determine what processor loads it was able to effectively schedule, and how various task sets affected the scheduler performance. These results could then be used to fine tune task sets for the desired performance. Lastly, because the scheduler is implemented on a functioning vehicle, real world events were simulated by constructing task sets of random length execution times.

Task sets were constructed of identical dummy processes consisting of C-language program headers combined with the OS-9 primitives required to send and receive signals. Data was gathered and sent to data files for later analysis in such a way as to not interfere with the execution times of the processes. Execution times were controlled by varying a timing loop in the individual processes.

B. SCHEDULER VERIFICATION

To verify scheduler performance, it was first necessary to show that the scheduler met the theoretical requirements of RMS. These requirements included pre-emption of lower priority processes by higher priority processes and a reliable means of

meeting process deadlines. The dummy tasks were modified in
such a way that the start and stop times of execution were
recorded. At the end of the sample run, these values were
written to a file for analysis by hand. The results of a
sample run with four tasks in the task set are shown in Figure
5.1 as a time-line similar to the time-lines found in [SHA
90].
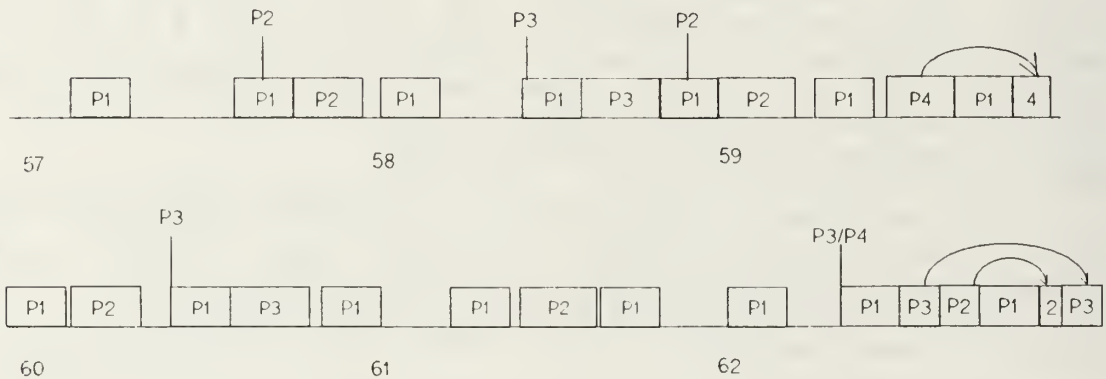
Process ID	Period (ticks)	Execution Time (ticks)
P1	40	15
P2	125	20
P3	200	25
P4	300	30

Figure 5.1 Time Line Displaying Preemptive Scheduling

Preemption can clearly be seen as process P1, the process with
the highest priority, forces the delay of all other processes
in the set until it has completed execution. Likewise, P2

36

delays all processes, save P1, until it has completed. It is also clear that all processes met their assigned periods. Although this example has only four tasks, it demonstrates that the scheduler meets the theoretical requirements of RMS. Other sample runs confirm these results for sets with a larger number of processes.

C. IMPROVEMENT OVER OS-9

OS-9 is advertised as a multi-tasking operating system, that supports real time operations [MICROWARE 87]. A test to show that a separate scheduler was needed to improve performance above that provided by OS-9 as implemented was required. The test consisted of two tasks sets, both with six processes running at 10 Hz. The first set was executed using OS-9 alone; the other using the scheduler software developed. Scheduler breakdown was measured as the percentage of missed deadlines. Missed deadlines were determined by comparing the number of signals sent vice the number of signals received by each process. All graphs have the percentage breakdown on the abscissa and the percentage of processor load on the mantissa. As shown in Figure 5.2, OS-9 began missing deadlines at processor loads of 30% while the implemented scheduler does not miss a deadline until the 90% point, a clear improvement. The poor performance of OS-9 was expected, as it implements no real-time structure, but, rather, the primitives used to construct those structures.

37
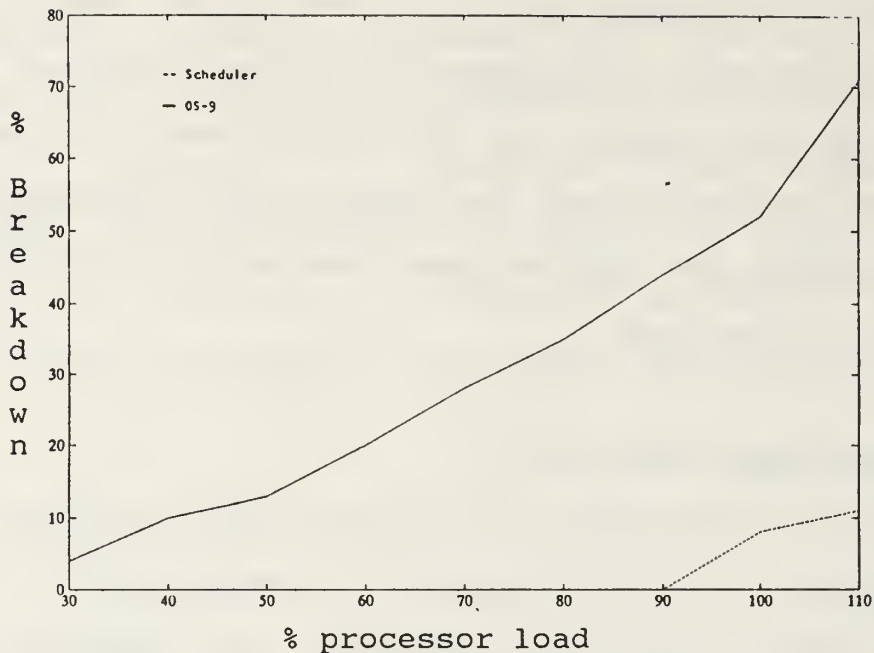
Figure 5.2 OS-9 versus RMS Performance

D. PROCESS LOAD VARIATIONS

In order to ensure an effective scheduler on-board the vehicle, experiments were conducted by varying task set parameters to test scheduler performance. Standardization between experiments was maintained by executing all task sets over ten cycles where a cycle being defined as the least common multiple of all periods in the task set.

1. Experiment 1

This experiment was designed to determine under what load the scheduler started to breakdown, or miss process deadlines. The task set consisted of six processes at a frequency of 10 Hz. Figure 5.3 shows that scheduler breakdown

38

began to occur at a 90% load, close to the maximum predicted by [SHA 90].
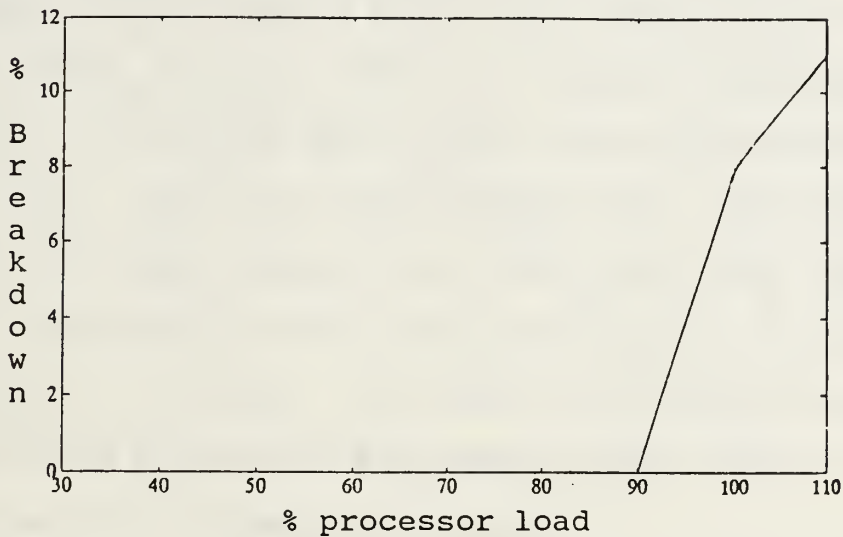
Figure 5.3 Performance for six 10 Hz processes

The processor load was varied by increasing or decreasing process execution times while maintaining constant process frequencies. Initial processor load was 50% and increased at 5% intervals upto 110%. Processor loads over 100% were included to study scheduler stability characteristics.

2. Experiment 2

This experiment replaced one of the 10 Hz processes in the first task set with a 20 Hz process. The sample run consisted of the same processor loads as the first experiment. As shown in Figure 5.4, the results were quite different from the first experiment.
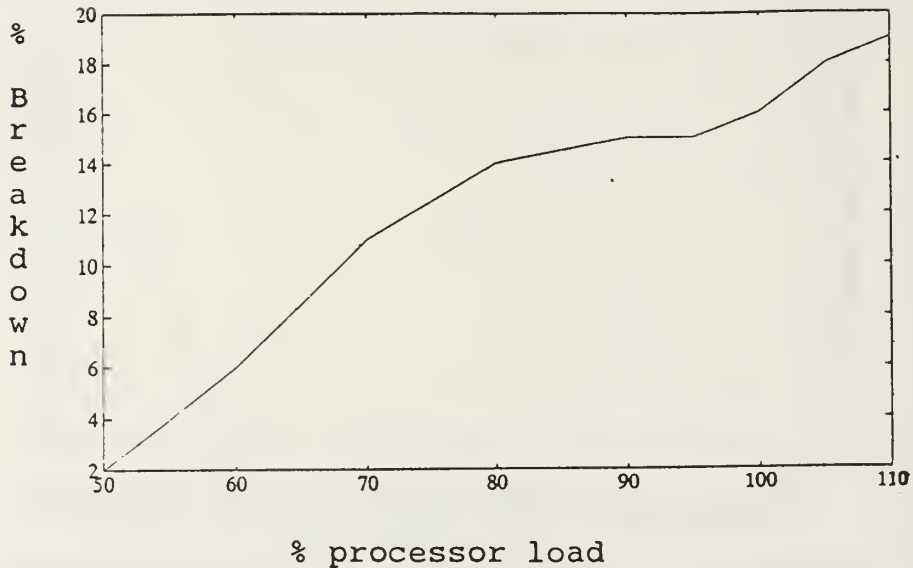
Figure 5.4 Performance for One 20 Hz Process and Five 10 Hz
Processes

The earlier break down was explained, theoretically, as being
due to the increased overhead caused by the increased number
of task swaps incurred by the higher frequency process. After
more experimentation, it was determined that the highest
practical frequency for a process is 10 Hz. If higher
frequencies are required, processor loads must be limited to
no more than approximately 50%.

3. Experiment 3

This experiment determined the number of lower
frequency processes that could be executed successfully.
Figure 5.5 shows four different task sets consisting of eight
tasks each, with the number of 10 Hz processes varying from

zero to three. The remaining processes are executed at 5 Hz.
Processor loads were the same as in the first two experiments.
It is interesting to note that while all task sets broke down
at approximately the 90% point, the breakdown was accelerated
proportional to the number of higher frequency processes
present. The various task groups are represented as follows:
solid line, eight 5 Hz tasks; dashed line, one 10 Hz task plus
seven 5 Hz tasks; dotted line, two 10 Hz tasks plus six 5 Hz
tasks; combination dotted and dashed line, three 10 Hz tasks
plus five 5 Hz tasks. From this it was determined that the
scheduler is more greatly affected by task frequencies than by
task execution times.
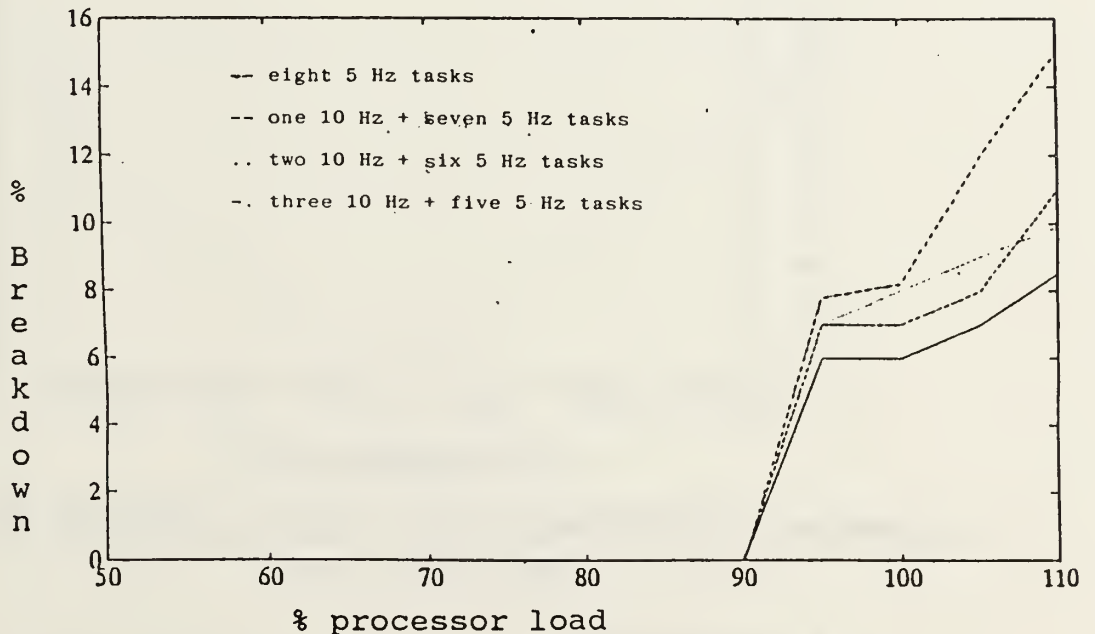
Figure 5.5 Performance with Different High Frequency
Processes

41

4. Experiment 4

This experiment was an attempt to devise and execute a 'typical' task set for the vehicle. A typical set being one that could be expected to be found on AUV-II during a normal mission. This set was constructed after discussion with project members who were developing software modules for guidance, control and sonar. The set consisted of eight processes; two at 10 HZ, four at 5 Hz, and two at 2 Hz. The results are plotted in Figure 5.6.
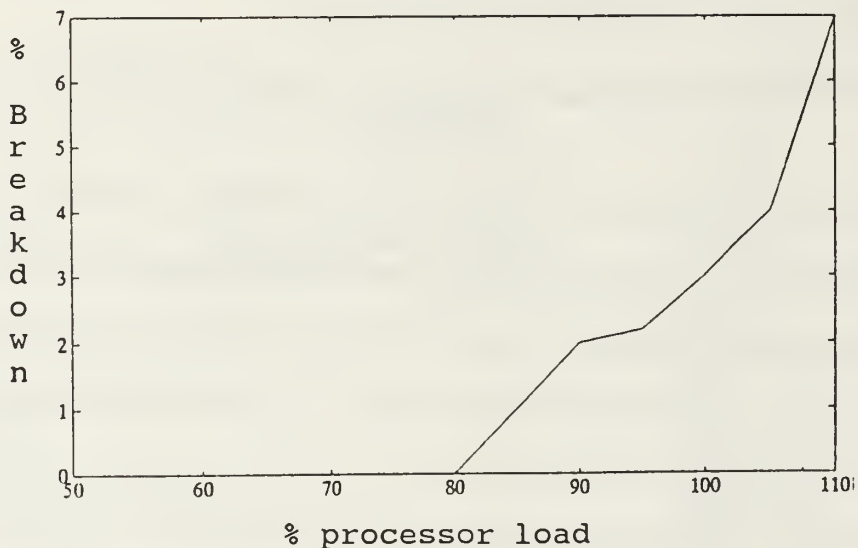
Figure 5.6 Performance for a Typical Task Set for AUV-II

5. Experiment 5

This experiment was an attempt at developing a task set that more realistically represented actual vehicle operation. The same task set was used as in experiment 4 with randomized execution times included. A random number generator was added

to each child process along with the logic needed to apply a normal distribution curve to the execution times. This code is given in Appendix C. The distribution was such that 50% of the executions were run at the mean time of execution; 40% of the executions were run at +/- 10% of the mean time; the remaining 10% of the executions were run at +/- 20% of the mean time. The lower utilization rate shown in Figure 5.7 was predictable. A process cannot 'save' excess allotted time from shorter than expected executions to make up for the longer than expected executions later encountered.

% processor load
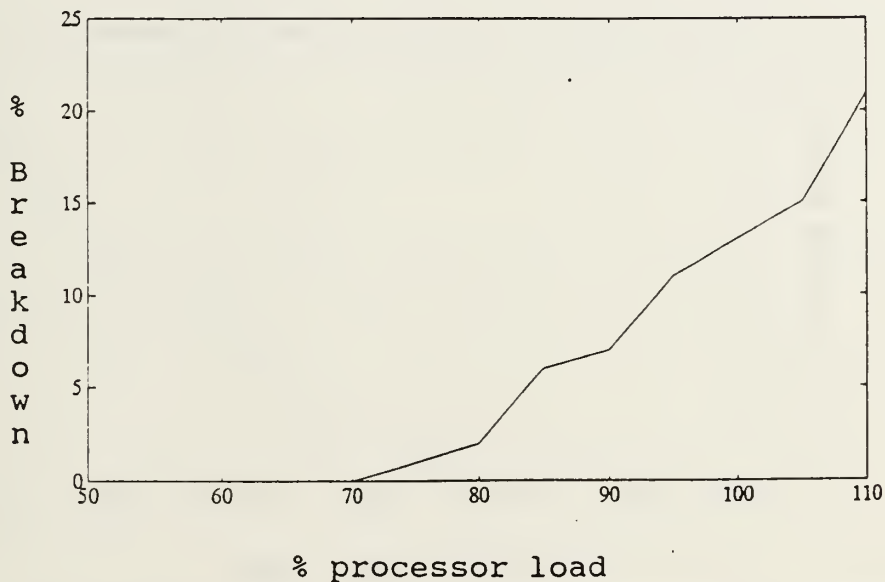
Figure 5.7 Performance for a Typical Task Set with Variable Execution Times

6. Experiment 6

This experiment was virtually the same as experiment 5 with slight variations in the periods and execution times. Figure 5.8 shows the results of the changed task set and is

included to show the breakdown qualities of the scheduler during sustained overload. The 'staircase' pattern was predicted prior to the experiment and was caused by preemption of the various lower priority processes. The more nearly vertical lines are caused by deadlines missed. When the graph first levels, all deadlines in the lowest priority process have been preempted, but all higher priority processes are still meeting their deadlines. The graph began to move upward again when the next lowest priority process began to miss deadlines.
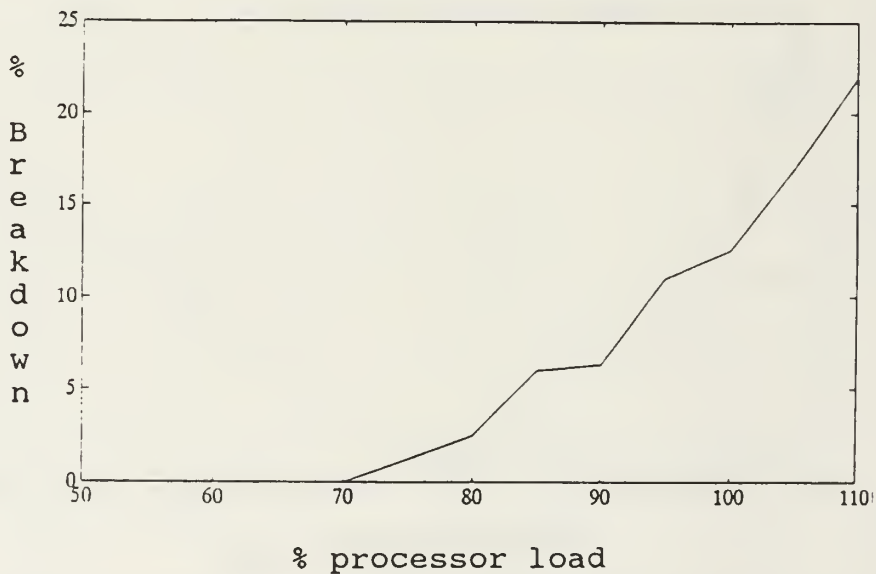
% processor load

Figure 5.8 Scheduler Stability Under Sustained Overload

This 'shedding' of lower priority processes would continue until only the highest priority process is still executing. This orderly break down ensures stability and allows for more efficient and safe vehicle operation under sustained overload

44

conditions.

The experiments reported are a representative subset of a much larger number of sample runs made to establish scheduler reliability and performance. As can be seen by these results the scheduler objectives listed in Chapter I have been met in all cases and the scheduler, as implemented, can be considered sophisticated enough for the present needs of AUV-II.

VI. CONCLUSIONS AND FUTURE RESEARCH

A. CONCLUSIONS

The primary objective of this research was to provide an efficient, hard real-time scheduler for the AUV that would be capable of maintaining a reasonably high processor utilization rate. This was to be accomplished over the varying group of task sets that the vehicle could be expected to encounter.

As implemented, the scheduler provides reliable, real-time scheduling with utilization rates above 80% for almost all variations of task sets. Under worst case conditions, and with varying task set execution times, a 75% utilization rate was not difficult to achieve. If care is used in developing the task set to be used, the system operator should have little trouble achieving utilization rates of 80% or higher.

Aperiodic events can also be handled with the current implementation, although not as smoothly as with periodic events.

As per the original specifications, the scheduler is stable under overload. Lower priority tasks are always suspended before higher priority tasks during overload conditions, so a subset of the most critical processes always meet their deadlines. Of course, the critical subset varies with processor load.

B. FUTURE RESEARCH

Although the scheduler meets all original specifications and should serve well into the future of the project, several modifications could be made to improve its performance and capabilities. These could include the following:

1. A more sophisticated approach to aperiodic events. The current method works well, but will not cover all possible eventualities. An alternative method was proposed in chapter V of this thesis.

2. Period transformation may be implemented to deal with long period processes that need a high priority.

3. A graphical interface could be included that allows for a running record of processor utilization, breakdown, and overload conditions.

4. A user-producer algorithm could be implemented to deal with the problem of dependent processes.

5. Once the appropriate compiler is available from GESPAC, the software should be translated from C to Ada to more accurately reflect Department of Defense policies.

6. Once Transputer boards are installed, modifications will be needed to allow the scheduler access to the increased resources available.

7. Finally, if an increase in utilization rates are required, a hybrid scheduling policy combining RMS and dynamic priority assignment may be implemented.

APPENDIX A. STARTUP MODULE SOURCE CODE

```c
/* Program: START_SCHED.C
   Purpose: To assign highest priority to scheduler
   Author:  LT Brent L. Leatherman  20 MAY 1991
*/

#include <stdio.h>
#include <errno.h>
#include <procid.h>

/* the following function are all OS-9 specific */

extern int wait(),os9forkc(),exit(),intercept();
extern char **environ;


/* the following is the structure required to fork a process
   the process to be forked MUST reside in the CMDS directory
   as a compiled, executable module -NOT LINKED- with any
   other module.
*/

char *arg1[]={
     "rate_mono", /* this line in the structure contains */
     0,};         /* the name of the process to be forked */

/* the following is the dummy intercept handler required when
forking a process */

icpthand(signum)
int signum;
  {
     /* intercept handler */
  }

main()
{    unsigned status; /* used in wait command */

     intercept(icpthand);
     /* following is process creation */
     os9exec(os9forkc,arg1[0],arg1,environ,0,62000,3)
     wait(&status); /* suspend process until signal is
                       received from scheduler */
     printf("Received signal from proc %d\n",status);
     /* completion message to sysop */
     exit(0); /* END PROGRAM */
}
```

```
/*      Program: RATE_MONO.C
        Purpose: To collect task set parameters, prioritize
                 the task set, determine task set
                 schedulability, and then to carry out the
                 scheduling policies.
        Author: LT Brent L. Leatherman 15 APRIL 91
        Theory: Based on rate monotonic scheduling theory
                [SHA 90]
*/


#include <stdio.h>
#include <math.h>
#include <errno.h>
#include <procid.h>
#include <setsys.h>
#include <signal.h>

/* constants */

    /*SIZE determines number of tasks in set */
#define SIZE 15

    /* MAX_PRIORITY determines most critical tasks priority */
#define MAX_PRIORITY 60000

/* The following functions are all OS-9 specific functions.
   Kill(),Exit() & Wait() all deal with signals
   OS9forkc(), Intercept() & Environ deal with process
                                    creation
   Sleep deals with process timing
   getpid() & _get_process_desc() deal with process
                                    characteristics
*/

extern
int
  kill(),os9forkc(),exit(),wait(),sleep(),intercept(),
  getpid(),_get_process_desc();
  extern char **environ;
```

```c
/*   This   structure   contains   the   required   process
characteristics needed for process creation. A blank line is
required to hold the name of each process that will be created
*/

char *arg1[]=
   {
      "           ",  /* each line is 10 characters long - the */
      "           ",  /* length of the name of the process with*/
      "           ",  /* no '.C' appended. The number of lines */
      "           ",  /* must equal the variable SIZE          */
      "           ",
      "           ",
      "           ",
      "           ",
      "           ",
      "           ",
      "           ",
      "           ",
      "           ",
      "           ",
      "           ",
      0,
   };

/* The same structure as arg1, used in the process of timing
individual processes  by measuring the execution time of a
modified process named the same as before with an 'a' appended
to the executable module */

char *arg2[]=
   {
      "                 ",
      "                 ",
      "                 ",
      "                 ",
      "                 ",
      "                 ",
      "                 ",
      "                 ",
      "                 ",
      "                 ",
      "                 ",
      "                 ",
      "                 ",
      "                 ",
      "                 ",
      0,
   };
```

50

```
/* the following procedure is a dummy intercept handler that
OS-9 requires when returing from a fork command */

icpthand(signum)
int signum;
 {
      /* intercept handler */
 }




main ()
{
   FILE *file; /* output file for signal recording */

/*  the  following  variables  are  the  various  loop  controls
required. The variable n, however, is the number of tasks in
the task set. /*

   int m,z,t,p,i,j,k,l,n,
       count,
       mask, /* used to mask off top end of time stamp */
       tick1, /* used to hold value of tick from time */
       tick2; /* holds second tick value for comparison */


   double T[SIZE], /* task set periods */
          C[SIZE], /* task execution times */
          W[SIZE][SIZE], /* holds intermediate values in
                          schedulability tests. See [SHA 86].
                                                         */
          Li[SIZE][SIZE], /* next set of intermediate values.
                                                         */
          Lint[SIZE], /* holds values when converting from
                          continous to discrete quantities */
          L; /* the final value, if <= 1.0, the set is
                 schedulable */
          sched_pt[SIZE*2], /* holds scheduling point values
                                arbitrarily twice the maximum
                                number of processes */
          temp,term1,temp1,temp2,chk; /* simple holding
                                          variables */
   int LCM, /* the least common multiple of all periods */
       process_pri[SIZE], /* process priorities */
       LCM_ARR[5*SIZE];/* used in the calculation of the LCM
                          arbitrarily set to 5 times the number
                          of processes        */
```

51

```c
    double SCOREBOARD[SIZE][2]; /* a table containing periods
                                   and pids for all processes */

    unsigned status,svalue; /* used by OS-9 functions */

/*  the  following  are  used  in  the  OS-9  timer  system  to
determine both elapsed and real-time */

    int start_sec,start_tick,adjust,date,
        time,tick,time1,time2,mask2,sec,index,start_time;
    short sigcode,day;

/* the following are used to determine process characteristics
*/

    int pid, /* process id # */
        gpiderr, /* error recorder from fork command */
        loop_con; /* signal control variable */
    procid parent;


    sigcode=0x01; /* hex signal used to wake a process */
    pid=getpid(); /* function to determine process ID */
    mask=0x0000ffff; /* hex code used to mask off top four
                        bytes of timer variable in order to
                        obtain number of ticks from system
                        clock */


    process_pri[0]=MAX_PRIORITY; /* set highest priority to
                                    the most critical task */
    file=fopen("rate.out", "w"); /* open file for output */
/****************************************************** */
/* The following section is input to the program
   primarily task execution time C[] & task periods T[] */
/* ****************************************************** */

    printf("Number of forks desired:\n"); /* controls the
            number of signals sent to the called processes.
            Strictly used in the lab. Onboard AUV-II, will need
            to be disabled or set to infinity  */



    scanf("%d",&loop_con);
    printf("\n\n");
    /* number of processes in task set */
    printf("Number of processes to schedule: ( <=15 )\n");
    scanf("%d",&n);

    /* following loop brings in task set parameters */
```

```c
for(i=0;i<n;i++)
    {
        printf("Enter the name of the process # %d\n",(i+1));
        scanf("%s",arg1[i]);
        for(j=0;j<9;j++)
                arg2[i][j]=arg1[i][j];
        strcat(arg2[i],"a");/* used for process timing */
        printf("Enter the period of process #
                %d(ticks)\n",(i+1));
        scanf("%F",&temp);
        /* OS-9 eccentricity - scanf commands MUST use
            capital F vice lower case f */
        T[i]=temp;
        printf("\n\n");
        printf("Enter the execution time\n");
        scanf("%F",&temp);
        C[i]=temp;

/* the following code (commented out), if implemented allows
the timing of the processes to be done by the system vice
human entry */

/*      _sysdate(3,&time,&date,&day,&tick); /* OS-9 timer
                                                call */
        printf("Starting second is %d\n",time);
        /* get start time of process creation */

        tick1=(tick & mask); /* getting time in ticks */
        time1=time*100+tick1; /* accounts for tick rollover
                                on the second */
        intercept(icpthand);/* dummy procedure needed prior
                                to fork */
        /* creation of a process */
        os9exec(os9forkc,arg2[i],arg2,environ,0,63000,3);
        wait(0);/* scheduler waits for completion of process
                                                        */
        /* get end time of process call */
        _sysdate(3,&time,&date,&day,&tick);
        tick2=(tick & mask);
        time2=(time*100+tick2);
        /* following is elapsed time calculation */
        C[i]=time2-time1;
        C[i]=C[i]-40;/* correction factor for fork overhead
                                                        */
        end of self timing sequence   */
        /* the following alerts sysop that timing is complete
            and reports the results of the timing */

        printf("**** Process # %d timing ****\n",(i+1));
        printf("execution time is: %.1f\n",C[i]);
        printf("**************************\n");
        printf("\n");
```

```
        }


/* **************************************************** */
/* The following code sorts the matricies C[] & T[] in
descending order based on process execution times ( C[] ).
This is done by an insertion sort routine. The sort is needed
later in the program for priority assignment based on process
execution times. */
/* **************************************************** */


for(i=1;i<=n;i++)
    {if (T[i]>T[i+1])
      {chk=0;
       temp1=C[i];
       temp2=T[i];
       for(j=(i-1);j>=1;j--)
           {C[j+1]=C[j];
            T[j+1]=T[j];
            if(T[j-1]<=temp2)
              {C[j]=temp1;
               T[j]=temp2;
               j=1;
               chk=1;
              }
           }
        if(chk!=1)
           {C[1]=C[0];
            T[1]=T[0];
            C[j]=temp1;
            T[j]=temp2;
           }
       }
    }


/* end sort routine */

/* **************************************************** */
/* the following code computes the scheduling points as
described in Chapter IV. The points are stored in the array
sched_pt[] */
/* **************************************************** */

     count=0;
     for(i=0;i<n;i++)
        {
           for(j=0;j<=i;j++)
             {
                for(k=1;k<=floor(T[i]/T[j]);k++)
                  {
```

54

```
                    flag=0;
                    S=k*T[j];
                    for(p=1;p<count;p++)
                        {if(S==sched_pt[p])
                            flag=1;
                        }

                    if(flag==0)
                      {
                       sched_pt[count]=S;
                       count++;
                      }
                }
            }
        }


/* *************************************************** */
/* The following code computes the Wi(t) as described in
   chapter IV of this thesis. */
/* *************************************************** */

    term1=0;
    for(i=0;i<n;i++)
        {
        for(t=0;t<count;t++)
            {
            for(j=0;j<i;j++)
                {
                    term1=term1+C[j]*ceil(sched_pt[t]/T[j]);
                }
            W[i][t]=term1;   /* Eqn (4.2) */
            term1=0;
            }
        }

/* *************************************************** */
/* The following code computes the Li term as described in
   equation 4.3 in chapter IV */
/* *************************************************** */

    for(i=0;i<n;i++)
        {
        for(t=0;t<count;t++)
            {
            Li[i][t]=(W[i][t]/sched_pt[t]); /* Eqn (4.3) */
            }
        }
```

```
/* ********************************************** */
/* The following code finds the minimum Li for (0<t<Ti) as
   described by eq. 4.4 of chapter IV. */
        /* **********************************************
*/

    for(i=0;i<n;i++)
        Lint[i]=10000;/* arbitarily large number*/

     for(i=0;i<n;i++)
       {
         for(t=0;t<count;t++)
           {
             if(Li[i][t]<Lint[i]) /* Eqn (4.4) */
                 Lint[i]=Li[i][t];
           }
       }


/* ************************************************** */
/* The following code computes the maximum value of Li. If
this  value  is  less  than,  or  equal  to 1 the task set  is
schedulable, else it can not be done. This routine will also
tell the user how much excess processor time is left */
/* ************************************************** */

   L=0;
   for(i=0;i<n;i++)
     {
       if(Lint[i]>L)
           L=Lint[i];
     }
    U=0;  /* determines processor load */
    for(i=0;i<n;i++)
       U=U+C[i]/T[i];/* if processor load > 1.0, not sched.*/
    if ((L<=1)&(U<=1)   /* Eqn 4.5 */
         printf("Process set schedulable.\n\n");
     if((L>1)||(U>1))
         printf("Process set not schedulable\n\n");
         printf("Total remaining processor time is:
                  %.2f",(1-U)*100);
         printf(" percent\n\n\n\n");


/* ************************************************** */
/* The following code computes the least common multiple of
the  periods  of  all  processes  in  the  task  set  in  order  to
calculate relative priorities. */
/* ************************************************** */
```

56

```
       temp=1;
       for(l=0;l<n;l++)
         temp_period[l]=T[l];
       for(l=0;l<n;l++) /* scan list for prime numbers */
       {
           /* check for even divisors using modulo arithmetic */
           for(k=0;k<n;k++)
             {
                 if((l!=k)&(temp_period[l]%temp_period[k]==0))
                   temp_period[l]=1; /* check for divisibility */
                 }
         }
       for(l=0;l<n;l++)
         {
           m=temp_period[l];
           count=0;
           for(i=2;i<=m;i++)
             {
               if((m%i)==0)
                 {
                   flag=0;
                   for(j=0;j<count;j++)
                     {
                       if(i==LCM_ARR[j])
                         flag=1;/* recording LCM points */
                     }
                   if(flag==0)
                     {
                         LCM_ARR[count]=i;
                         count++;/* how many LCM points */
                     }
                   m=m/i;
                 i=2;
                 }
             }
         for(k=0;k<count;k++)
           {
             temp=temp*LCM_ARR[k];
             LCM_ARR[k]=0;
           }
       }
       LCM=temp;

/* end of LCM routine */


/* ****************************************************** */
/* The following code assigns priorities to the scheduled
processes. The processes with the shortest execution time have
the highest priorities as per [SHA 90]. The exact priority is
assigned in accordance with an equation (4.1) */
/* ****************************************************** */
```

```c
      temp=1;
      temp1=0;
      temp2=1;
      for(i=1;i<n;i++)
          {
           for(j=0;j<=(i-1);j++)
              temp2=temp2*T[j];
           for(k=0;k<=(i-1);k++)
              {
               for(l=0;l<=(i-1);l++)
                 {
                  if(l!=k)
                     temp=temp*T[l];
                 }
               temp1=temp1+C[k]*temp;
               temp=1;
              }



           process_pri[i]=process_pri[0]-
                        ceil(LCM*(temp1/temp2)*5*i);
                             /* Eqn (4.1) */
           temp1=0;
           temp2=1;
          }
      temp1=0;
      for(i=0;i<n;i++)
          {if(T[i]>temp1)
             temp1=T[i];
          }
       for(i=1;i<n;i++)
           process_pri[i]=process_pri[0]-(i*temp1);


/* end of priority assignment routine */

/* following code prints statistical messages to a file on
disk - a compilation of task set parameters */

fprintf(file,"process#\texec. time\t  period\t\tpriority\n");
fprintf(file,"_____\t_____\t _____\t\t_____\n");

      for(i=0;i<n;i++)
         {fprintf(file,"    %d",(i+1));
          fprintf(file,"            %.1f",C[i]);
          fprintf(file,"            %.1f",T[i]);
          fprintf(file,"            %d\n",process_pri[i]);
         }
fprintf(file,"\n\n\n\n\n\n");
```

58

```c
/* ****************************************** */
/* the following loop is an infinite while loop that will
   control the processes to be forked off. A "scoreboard"
   type approach used in that when a process is signaled,
   it's next execution time is computed and scheduled. Psuedo-
   code for this procedure may be found in figure 4.3. */
/* ****************************************** */

   count=0;
   _sysdate(3,&time,&date,&day,&tick); /* get time */
   tick1=(tick & mask);
   pid=getpid(); /* get process id number */
   adjust=0;
   for(i=0;i<n;i++)
     {
      intercept(icpthand); /* dummy intercept handler */
   os9exec(os9forkc,arg1[i],arg1,environ,0,process_pri[i],3);
      _get_process_desc(pid, sizeof parent, &parent);

      SCOREBOARD[i][0]=T[i]; /* enter original period in
                                  table */
      SCOREBOARD[i][1]=parent._cid; /* enter process id */
      printf("process id # %d is %d\n",(i+1),parent._cid);
      printf("rate_mono pid is: %d\n",pid);
    }

   while(count < loop_con) /* infinte loop when in vehicle */
     {
         sec=0;
           for(i=0;i<n;i++)
             {if(SCOREBOARD[i][0]==0)
                { INDEX[sec]=i; /* record values */
                  sec++;
                }
                /* ******************************** */
                /* find all processes with zero time */
                /* left. This indicates that they    */
                /* are ready for execution. These    */
                /* are recorded in INDEX.            */
                /* ******************************** */
             }
       _sysdate(3,&time,&date,&day,&tick);
       tick1=(tick & mask); /* start time */
       for(z=0;z<sec;z++)
         {
           sigcode=SCOREBOARD[INDEX[z]][1]; /* get process id
                                    that is to be signaled */
           kill(sigcode,1); /* send wake up signal */
           count++; /* counts number of signals sent */
         }
       _sysdate(3,&time,&date,&day,&tick);
       tick2=(tick & mask); /* end time */
```

```
      if(tick2<tick1)
        tick2=tick2+100;
      adjust=tick2-tick1; /* time used in signaling process
                                                          */
/* the following code reassigns original period to signaled
   processes */

   for(i=0;i<sec;i++)
        SCOREBOARD[INDEX[i]][0]=T[INDEX[i]];

/* the following code subtracts the time used in signaling
   from all processes */

   for(i=0;i<n;i++)
        SCOREBOARD[i][0]=SCOREBOARD[i][0]-(adjust-1);

      temp=10000;
      for(i=0;i<n;i++)   /* find shortest time to execution
                                                          */
        if(SCOREBOARD[i][0]<temp)
          temp=SCOREBOARD[i][0];

    for(i=0;i<n;i++) /* subtract smallest time */
                     /* from all periods /*
        SCOREBOARD[i][0]=SCOREBOARD[i][0]-temp;

/* ***************************************************** */
/* the next code puts 'main' to sleep for the shortest
   period remaining in the table. */
/* ***************************************************** */

      if(temp <=0)
        fprintf(file,"problem with temp\n");

    /* OS-9 eccentricity. If tsleep is used with the value 0
       (zero) or below, it goes to sleep forever, system
       reboot is required to restart system. */

      if(temp>0)
          {svalue=ceil(temp);
           tsleep(svalue);
            /* system suspension to allow execution of
               signaled processes */
          }
    }

/* **************************** */
/* end of fork coding */
/***************************** */


/* the following code allows for an orderly, gracful end to
```

60

the scheduler. It kills all active processes, returning all
resources back to the system manager. */

```
    fclose(file);
    tsleep(1000);
    for(i=0;i<n;i++)
      {sigcode=SCOREBOARD[i][1];
       kill(sigcode,0);
       }
    exit(pid);
/* END PROGRAM */
}
```

APPENDIX C. MODIFICATIONS TO CALLED MODULES

```c
/* Program: PROCi.C
   Purpose: To illustrate a dummy process used in
            experimental verification of the scheduler,
            and the modifications necessary to function with
            the implemented scheduler.
   Author:  LT Brent L. Leatherman 12 MARCH 1991


#include <stdio.h>
#include <errno.h>
#include <signal.h>
#include <setsys.h>

/* following functions are OS-9 specific */

extern int exit();
extern char **environ;

main()
  {
    FILE *file; /* needed for statistical dump following
                   execution */
    int date,time,tick,mask; /* needed to access system
                                clock */
    short day;
    int z,k,i; /* loop control variables */
    int hold_time[100][4]; /* holds start and stop times */
    int count;
    mask=0x0000ffff; /* mask used to get time in ticks */
    count=0; /* counts number of signals received */
    z=1;

/* The following is the infinite loop modification required
   by the scheduler. It needs to encompass all code that is
   to be executed each time and NOT initialization commands
   that are to be done only once. If the self-timed module
   execution option is being used, this loop should be
   removed so that an accurate timing of one execution is
   recorded. */

    while(1)
      {
      printf("proc1 count = %d\n", count); /* msg to sysop */
      _sysdate(3,&time,&date,&day,&tick); /* get start time */
      tick=(tick & mask);
      hold_time[count][0]=time; /* record start time in
                                   seconds */
      hold_time[count][1]=tick; /* record start time in ticks
                                                            */
```

```c
    /* The following is a delay loop used to simulate process
       execution times, replaced by process code on AUV-II . A
       loop count of 3300 equals a time period of one tick, or
       one millisecond. */

    for(i=1;i<3300;i++)
         z=z+1;
      _sysdate(3,&time,&date,&day,&tick);
      tick=(tick & mask); /* find and record finish times */
      hold_time[count][2]=time;
      hold_time[count][3]=tick;
      count++;

/* The following code is needed due to the measurable time
   required to write to a hard disk drive. Therefore,
   instead of writing each time as it comes in, the times
   are recorded and then written in a batch at the users
   discretion using modulo arithmetic on the variable count.
   In this example, times are recorded every 20 signals */

    if((count%20)==0)
      /* the following are the statistical outputs */
      {
          file=fopen("out1","a");
          fprintf(file,"stats for process #2\n");
          for(z=0;z<count;z++)
          {
            fprintf(file,"proc 2 start time:
             %d.%d\n",hold_time[z][0],hold_time[z][1]);
            fprintf(file,"proc 2 end    time:
             %d.%d\n",hold_time[z][2],hold_time[z][3]);
          }
        fclose(file);
        count=0;
      }
    }
    exit(0);
/* END PROGRAM */
}
```

```
/* Program: PROCj.C
   Purpose: To demonstrate the random number generator added
            to vary execution time. The rest is identical to
            the first dummy procedure.
   Author:  LT Brent L. Leatherman 12 MARCH 1991
*/

#include <stdio.h>
#include <errno.h>
#include <signal.h>
#include <setsys.h>

extern int exit();
extern char **environ;

main()
  {
    FILE *file;
    int date,time,tick,mask;
    short day;
    int z,k,i;
    int hold_time[100][4];
    int count,next,
        rand_num,
        length, /* nominal execution time */
        length1, /* execution time -20% */
        length2, /* execution time -10% */
        length3, /* execution time +10% */
        length4; /* execution time +20% */
    mask=0x0000ffff;
    count=0;
    z=1;
    _sysdate(3,&time,&date,&day,&tick);
    next=(tick & mask);
    length1=2666; /* various execution times */
    length2=3000;
    length3=3666;
    length4=4000;
  while(1)
     {
    length=3333; /* nominal execution time */

/* random number generator */
/* initial seed comes from system clock */

    next=next*1103515245+12345;
    rand_num=((next/65536) % 32768);
```

```c
/* The following code sets up the distribution pattern, as
   follows:

        5% of the time execution time is -20%
       20% of the time execution time is -10%
       50% of the time execution is nominal
       20% of the time execution time is +10%
        5% of the time execution time is +20%

*/

    if(rand_num<0)
        rand_num=rand_num*(-1);
    if(rand_num<1638)
      length=length1;
    if((rand_num>=1638) & (rand_num<8191))
      length=length2;
    if((rand_num>=24574) & (rand_num<31127))
      length=length3;
    if(rand_num>=31127)
      length=length4;
    printf("proc10 count = %d\n", count);
    _sysdate(3,&time,&date,&day,&tick);
    tick=(tick & mask);
    hold_time[count][0]=time;
    hold_time[count][1]=tick;
    for(i=1;i<length;i++)
        z=z+1;
    _sysdate(3,&time,&date,&day,&tick);
    tick=(tick & mask);
    hold_time[count][2]=time;
    hold_time[count][3]=tick;
    count++;
      if((count%20)==0)
    {
          file=fopen("out1","a");
          fprintf(file,"stats for process #2\n");
        for(z=0;z<count;z++)
          {
              fprintf(file,"proc    2    start    time:
%d.%d\n",hold_time[z][0],hold_time[z][1]);
              fprintf(file,"proc    2    end          time:
%d.%d\n",hold_time[z][2],hold_time[z][3]);
          }
          fclose(file);
          count=0;
      }


    }
    exit(0);
```

65

APPENDIX D. OS-9 SPECIFIC REAL-TIME PRIMITIVES

The following is a list of real-time primitives supported by the OS-9 operating system. All descriptions come from [MICROWARE 87].

- os9forkc: used to create a process
- exit: used to terminate a process
- wait: used to suspend process until a signal is received
- sleep: used to suspend a process for a specified number of seconds
- tsleep: used to suspend a process for a specified number of ticks
- alm_atdate: used to send signal at specified Gregorian date
- alm_atjul: used to send signal at specified Julian date
- alm_cycle: used to send signals at constant intervals
- alm_set: used to send a signal after a specified elapsed time
- kill: used to delete a process
- abort: used to stop process from keyboard
- wake: used to place suspended process on ready to execute queue

LIST OF REFERENCES

[Bihari 90]
    Bihari, T.E., "Comments on the computer software and hardware
    of the AUV-II," paper presented to the Computer Science
    Department, Naval Postgraduate School, Monterey, Ca., July
    1990.

[Cloutier 90]
    Cloutier, M., Real-Time Control Software for the Autonomous
    Underwater Vehicle, Master's Thesis, Naval Postgraduate
    School, Monterey, Ca., June 1990.

[Dibble 88]
    Dibble, P., OS-9 Insights - An Advanced Programmers Guide to
    OS9/68000, Microware Systems Corporation, 1988.

[GESPAC 88]
    GESMOS-68 OS9/68000 Operating System, Ver. 2.3, GESPAC,
    Inc.,Geneva, SA, 1988.

[GESPAC 89]
    GESPAC 1989 - Boards, Systems, Software, Product Catalog,
    GESPAC, Inc., Geneva, SA, 1989.

[GRiD 88]
    GRiD Systems Corp., GRiDCASE 1535 EXP Owner's Guide(1535-40),
    Rev. A, Fremont, CA, November 1988.

[Healy 90]
    Healy, A.J., McGhee, R.B., Cristi, R., Papoulis, F.A., Kwak,
    S.H., Kanayama, Y., Lee, Y., "Mission Planning, Execution,
    and Data Analysis for the NPS AUV II Autonomous Underwater
    Vehicle," Proc. of International Advanced Conference and
    Exposition, 1st Workshop on Mobile Robots for Subsea
    Environments, Monterey, Ca., October 23-26, 1990, pp 177-186.

[Lehoczky 89]
    Lehoczky, J.P., Sha L., and Ding, Y., "The Rate Monotonic
    Scheduling Algorithm - Exact Characterization and Average
    Case Behavior," Proc. IEEE Real-Time Systems Symp., CS Press,
    Los Alamitos, Ca., Order No. 2004, pp. 166-171, 1989.

[Levi 90]
    Levi, S.T., and Agrawala, A.K., Real Time Systems Design,
    McGraw-Hill Publishing Company, 1990.

[Liu 73]
    Liu, C.L., and Layland, J.W., "Scheduling Algorithms for
    Multiprogramming in a Hard Real-Time Environment," J. ACM,
    Vol. 20, No. 1, pp. 46-61, 1973.

[Microware 87]
    Microware Systems Corp., Using Professional OS9, Ver. 2.3,
    1987.

[Sha 90]
    Sha, L. and Goodenough, J.B., "Real-Time Scheduling Theory
   and Ada," Computer, pp. 53-62, April 1990.

[Sha 86]
    Sha, L., Lehoczky, J.P., and Rajkumar, R. "Solutions for some
    Practical Problems in Prioritized Preemptive Scheduling,"
    Proc. IEEE Real-Time Systems Symp., CS Press, Los Alamitos,
    Ca., Order No. 749, pp. 181-191, 1986.

[Shukla 91]
    Conversation between Shridhar Shukla, Code 32, Naval
    Postgraduate School, and the author, 27 June 1991.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center   2
    Cameron Station
    Alexandria, Virginia 22304-6145

2. Library, Code 52   2
    Naval Postgraduate School
    Monterey, California 93943-5002

3. Department Chairman, Code EC   1
    Department of Electrical and Computer Engineering
    Naval Postgraduate School
    Monterey, California 93943-5000

4. Professor Anthony Healy, Code ME/Hy   1
    Department of Mechanical Engineering
    Naval Postgraduate School
    Monterey, California 93943-5000

5. Professor Robert McGhee, Code CS/Mz   1
    Department of Computer Science
    Naval Postgraduate School
    Monterey, California 93943-5000

6. Professor Shridhar Shukla, Code EC/Sh   2
    Naval Postgraduate School
    Monterey, California 93943-5000

7. Professor Roberto Cristi, Code EC/Cx   2
    Naval Postgraduate School
    Monterey, Califorina 93943-5000

8. LT Brent L. Leatherman   2
    108 Greenway Drive
    Goshen, Indiana 46526