Operating System Simulator Project

1. Introduction

This project is designed to be completed within approximately 10 weeks. You will learn much from this project if the proper time and thought is invested. Some of the learning objectives and goals are the following:

(a) to learn and implement some of the basic concepts of event-driven simulation;
(b) to learn fundamental issues of resource allocation and management in multiprogramming operating systems;
(c) to learn and implement the mechanism of context switching and interrupt handling;
(d) to learn and implement the basic flow of control within an operating system;
(e) to learn and implement different resource management algorithms;
(f) to learn and implement fundamental data structures of an operating system;
(g) to develop and practice good programming and debugging habits;
(h) to gain some experience implementing a complex program;
(i) to develop and sharpen your skills as a C programmer;

The value of this project as a learning experience is directly proportional to the time you give it. It will be very time consuming, there is no mistake about that. However, this project can give you an edge when you enter the work force after graduation - it should be documented on your resume'. I'll be glad to give you some pointers if you wish.

2. Project Overview.

The program you will write (hereafter called the "simulator") is designed to simulate the action of both hardware and software components of a simple time-sharing computing system. Your simulator will be organized so that C functions will model the behavior of hardware components as well as components of the operating system itself. In particular, simulator components will be written to model CPU and MEMORY hardware, interrupt handlers, the CPU scheduler, and process management functions of the system. Other components of the simulator will be provided to you at the beginning of the project.

One of the input files (CONFIG.DAT) to the simulator contains data describing the exact configuration of the system being simulated; for example, the number of interactive terminals, the number and characteristics of peripheral devices, the speed of the CPU, and the size of

MEMORY, to name a few. The part of the simulator designed to read this file and initialize the simulation environment will be given to you at the outset.

Several other input files are needed to run the simulator. The LOGON.DAT file contains a description of user logon events by specifying the time they will occur and the ids of users logging on. To model the activity or behavior of each interactive user, a file called SCRIPT.DAT is input to the simulator. This file contains a "process script" identifying the sequence of programs run by each user during an interactive session. To model the behavior of each program designated in the process script, a "program script" must be input. There are five types of programs your simulator will model: EDITOR, PRINT Spooler, COMPILER, LINKER, and arbitrary USER programs. For each of these program types a file will be input containing a script for each program "instance" of that type; eg, the EDITOR may be executed a total of five times by all users, therefore five EDITOR scripts will be input via EDITOR.DAT. Details on these files will be described later.

Program scripts will be "executed" by your CPU module to generate, in time sequence, the SVC calls made by that program. These calls become events processed by the interrupt hardware placing the operating system in execution under control of the Interrupt Handler(another function you will write). The Interrupt Handler calls the appropriate service routine, which may call the Scheduler, etc. Finally, when interrupt service is complete, control returns to the interrupted program and "execution" of that program continues until it terminates via an END SVC.

Other types of events can occur during simulation: EIO events generated by some device, and TIMER events generated by the system interval timer. These events also trigger the interrupt servicing mechanism described above.

Simulation terminates when all interactive users have "logged off" and all I/O events have been serviced. A user "logs off" when the corresponding process reaches the end of its process script. Thus, when all process scripts have been exhausted, and all I/O activity ceases, the simulator can terminate.

Output from the simulator will be to a single file called "name.OUT", where "name" is your last name (truncated to 8 characters). One of the CONFIG parameters is the name of your output file. In addition to a summary of all CONFIG.DAT parameters read as inputs, your output file will contain a log of all events processed during simulation. Furthermore, images of loaded process and program scripts will be echoed to this file. Finally, debug output and simulation statistics will be written to this file.

As an aid during the development of your simulator, you will have access to the "instructor's version" called OSSIM. OSSIM will be available on the LAN. Running or using OSSIM can be beneficial in two ways:

(1) it provides a guide and a benchmark for your output;
(2) it provides a working example to enhance your understanding of the dynamics of the simulation.

You should learn to use OSSIM as soon as possible.

3. The System Model.

The system being simulated is a single processor, time-sharing system with the following hardware and software components.

Hardware: N interactive terminals. One central processor. Processor memory. An arbitrary number of peripheral devices.

Software: Interrupt handlers for LOGON, I/O completion, I/O-wait SVCs, I/O-Request SVCs, Program-end SVCs, Program-Abends; Memory manager; Loader; Scheduler; Editor; Compiler; Linker; and arbitrary user programs.

All input parameters describing the system being simulated are entered through the file CONFIG.DAT.

4. Simulator Specification

The simulation begins by processing events. The first event that occurs is an interrupt from a user terminal signaling a request to logon. The interrupt hardware changes the CPU and MEMORY states and gives control to the Interrupt Handler(IH). The IH examines the source and cause of the interrupt and calls the Logon Service routine.

The Logon Service routine creates a process control block (PCB) for the new terminal user and reads a process script from SCRIPT.DAT. It then allocates and loads the first program in the user's process script. Next, the service routine places the PCB in the CPU ready queue and signals the scheduler. When the very first logon event is serviced, the CPU will be idle. Consequently, the scheduler will assign the newly created PCB to the CPU. Then, the Dispatcher is called to give control to the new process. The Dispatcher prepares the program for execution and calls the CPU. The CPU interprets instructions contained in the program script until it encounters the next SVC call. It then creates an event corresponding to the SVC request and adds it to the event list. The CPU then terminates releasing control to the interrupt hardware to service the next hardware event.

The servicing of other events (interrupts) is similar. For example, when an SVC to start an I/O operation is serviced, an I/O request block (IORB) is allocated and queued for the requested device. An attempt is then made to start a new operation on the requested device. If the device is not busy, the next waiting IORB is de-queued and its request is initiated. The device operation is simulated by simply computing how long the I/O will take (based on byte count and device speed). The computed duration of the I/O transfer is then used to create an EIO event for the device which is added to the event list.

As you can see, the servicing of one event creates new future events. Eventually, as programs reach their end and I/O devices complete their requests, the event list will empty and the simulator will terminate.

The remainder of this section serves not only to amplify on the details of the simulation highlighted above, but to serve as a specification for the program you are to implement. Section 4.1 introduces the notion of Process Script, the model of how interactive users behave. Section 4.2 follows with the definition and discussion of Program Script, our model of how a typical program behaves from the Operating System's point of view. Section 4.3 describes the input files to the simulator, while Section 4.4 describes the output file.

4.1 The Process Model

The behavior of every terminal user must adhere to the process model,although the number and sequence of programs executed by each user will vary. A "process script" is any sequence of programs defined by the regular expression given in (1) that conforms to the transitions of the process model. It defines the system and user programs an interactive user will run during an interactive session. Each program specified in the process script must be allocated memory, loaded and run. The process script, in effect, defines the work load for the operating system determined by a given user.

(1)logon {editor, user, linker, compiler, printer}* logoff

Examples of valid process scripts:

Example 1: LOGON, EDITOR, PRINTER, LOGOFF

Example 2: LOGON, EDITOR, COMPILER, LINKER, USER, LOGOFF

Example 3: LOGON, PRINTER, EDITOR, PRINTER, LOGOFF

Process scripts are input via the file SCRIPT.DAT. Each time a user logon event occurs, a complete script is read from this file. This script defines the behavior of that user during the simulation. Consequently, the number of process scripts in this file must equal the number of LOGON events. See Section 4.3 for a more detailed description of SCRIPT.DAT and LOGON.DAT.

4.2 The Program Model

Each program in a process script will have a behavior defined by a "program model" for the corresponding program type. A separate model COULD be provided for the editor, compiler, linker, loader, printer, and user program classes. Each program model produces a "program script" consisting of a sequence of instructions from the set {SIO, dev, WIO, REQ, END }. A detailed explanation of these instructions will be given later in this section. The sequence of instructions in every program script must belong to the set generated by the grammar in figure 1, and must satisfy certain "additional constraints".

```
<SCRIPT>     --> <BODY> END

<BODY>       --> <SIO-PAIR> <BODY>
             --> <JMP-PAIR> <BODY>
             --> <SIO-PAIR> <BODY> <WIO-PAIR>
             --> <BODY> <BODY>
             --> null

<SIO-PAIR>   --> SIO  dev
<WIO-PAIR>   --> WIO  REQ
<JMP-PAIR>   --> SKIP JUMP
```

Additional Constraints:

- An REQ instruction must address a previous dev instruction.
- No two REQ instructions may address the same dev instruction.
- It is not necessary to have an REQ instruction for every dev instruction.

- A "dev" instruction must identify a valid device.

[Figure 1. Program Script Grammar and Additional Constraints.]

When a program script is executed, a sequence of events is generated corresponding to three types of SVC calls: request to Start I/O (SIO), request to Wait for I/O (WIO), and a request to terminate (END). In addition, a program may abnormally terminate due to some kind of error. For the purposes of this project, we will simulate only abnormal termination due to Memory addressing faults.

A detailed description of the instructions types used in the generation of program scripts is given below.

```
                          -------------
INSTRUCTION: SIO         |  SIO  |  t  |
                          -------------
```

SIO - Defines a CPU burst that ends with an SVC request to start an I/O operation. The I/O operation is described by the next dev instruction. SIO instructions differ from WIO instructions by permitting the current job to continue processing rather than possibly being blocked, and having to wait.

t - Defines the length of the CPU burst in instruction cycles.

```
                           --------------
INSTRUCTION: dev          |  dev | bytes |
                           --------------
```

dev - Defines the device to be used in the specified I/O operation. It must be the 4-character id of a device specified in the system device table.

bytes - Defines the byte count of the data transferred. Given the speed of the device, the time required for the I/O operation may be computed using the byte count.

```
                           --------------
INSTRUCTION: WIO          |  WIO  |  t   |
                           --------------
```

WIO - Defines a CPU burst that ends with a request to wait for an I/O operation identified by the REQ instruction that follows. This means the program can not continue until the identified I/O has operation has completed. If the I/O operation completes before a WIO is made, the program may continue processing; otherwise, the program must be blocked and removed from the CPU.

```
  t    - defines the length of the CPU burst.

                              --------------
INSTRUCTION: REQ             |  REQ | addr  |
                              --------------
```

```
  REQ  - Identifies the dev instruction of an I/O operation that must be
         completed before the program can continue. REQ instructions
         always appear immediately following WIO instructions.

 addr  - Is the logical address (segment, offset) of a dev instruction.
         This device instruction address must uniquely identify the
         operation to the system.


                              --------------
INSTRUCTION: SKIP            | SKIP |   n   |
                              --------------

 SKIP  - This instruction is used to build conditional branching and
         iteration.  The operand denotes a "skip count".  If the skip
         count is positive, its value is decremented and the next
         instruction is skipped.  If the skip count is zero, it remains
         unchanged and the next instruction is executed.  For our pur-
         poses, SKIP instructions will always be followed by a JUMP
         instruction.

 n     - The skip count (unsigned int).


                              --------------
INSTRUCTION: JUMP            | JUMP |  addr |
                              --------------

 JUMP  - This instruction is an unconditional transfer of control (jump)
         to the location specified by the logical "addr".

 addr  - Specifies the logical transfer address (segment, offset).

                              --------------
INSTRUCTION: END             |  END |   t   |
                              --------------

 END   - Defines the last CPU burst in a normal program execution.

 t     - Defines the length of that CPU burst in instruction cycles.
```

An example of a valid program script and a brief explanation of the meaning of the program instructions is given below.

Example 4: An EDITOR program script.

ADDRESS	INSTRUCTION	MEANING
[0,0]	SIO 5	The program begins by executing 5 instructions followed by an SVC interrupt to start an I/O operation and continue processing.
[0,1]	DISK 500	This is a dev instruction specifying the disk as the device on which to initiate I/O - 500 bytes of data are to be transferred.
[0,2]	SIO 20	The program continues executing for 20 instructions and then requests another I/O operation. Note the total program execution time is 25 CPU cycles at this point.
[0,3]	PRNT 100	100 bytes of information are to be transferred to the printer.
[0,4]	WIO 0	The program needs to wait for an earlier I/O operation to complete before it may continue processing - 0 specifies the number of CPU cycles the program executes before making the request to wait (in this case, none).
[0,5]	REQ [0,1]	The REQ instruction addr specifies the program is waiting for the operation initiated by the dev instruction at address [0,1] which is a disk operation. If the I/O has completed, the program may continue execution, otherwise it must be blocked (and the CPU will be rescheduled).
[0,6]	WIO 15	The program executes for 15 CPU cycles (bringing the total to 40), then requests to wait for I/O.
[0,7]	REQ [0,3]	The I/O being requested was initiated by the dev instruction at logical address [0,3] - it is the printer operation.
[0,8]	SIO 10	The program executes for 10 CPU cycles and then produces a request to initiate I/O - notice the program will end without ever requesting to wait for this operation.

```
[0,9]   PRNT   200     200 bytes are to be transferred to the printer.

[0,10]  END      5     The program will execute for 5 CPU cycles and
                       then terminates normally.
```

Example 5: A program script illustrating loops and conditionals.

```
ADDRESS INSTRUCTION            MEANING
------- -------------          -------
[0,0]   SKIP        5          Start loop with 5 iterations.
[0,1]   JUMP    [0,8]          When skip count 0, exit the loop.
[0,2]   SIO         5          Start I/O on DISK.
[0,3]   DISK      128
[0,4]   WIO        25          Wait for Disk to complete.
[0,5]   REQ     [0,3]
[0,6]   SKIP        0          Execute next instruction.
[0,7]   JUMP    [0,0]          Jump to start of loop.
[0,8]   END       200
```

4.3 Input Files.

The following input files are required by the simulator. The format and content of each is described below.

CONFIG.DAT: This file contains system configuration parameters, scheduling parameters, and debug options. Your simulator will only have to support the following parameters.

```
    LNAME=         Ignore this field and don't edit this line!!!
    DEVICES=       No. of devices (size of device table).  Immediately
                   following this parameter are the device descriptions;
                   one pair of parameters for each device.

    ID= aaaa  RATE= dddddd -+  Device Id must be 4 alphanumeric chars.
    ID= bbbb  RATE= dddddd  |  Device Rate must be in bytes/second.
        ..............      |
    ID= xxxx  RATE= dddddd -+

    TIME=          Defines the relative time units associated with LOGON
                   events.  The possible values are: NSEC,mSEC,MSEC,SEC.
    TERMINALS=     Maximum no. of interactive users.
    MEMSIZE=       Memory size in words.
    CPURATE=       Speed of CPU in nanoseconds/instruction
```

```
MAXSCRIPT=     Maximum length of a process script, including LOGOFF.
MAXSEGMENTS=   Maximum number of program segments.
SCHED=         Scheduling algorithm: FCFS, SJN, HPRN, RNDRBN
1/BETA=        Scheduling parameter for SJN and HPRN
               (ave. CPU service burst in no. of instructions)
RHO=           Smoothing factor for SJN or HPRN (0.0 <= RHO <= 1.0)
RRQUANTUM=     Time slice for Round Robin scheduling.
DEBUG_EVTQ=    --+
DEBUG_MEM=       |  Debugging flags: ON or OFF
DEBUG_PCB=       |
DEBUG_RBLIST=    |
DEBUG_RBQ=       |
DEBUG_CPUQ=    --+
```

Each parameter should be specified on a separate line in exactly the form shown above. At least one space should separate the parameter name from its value. The order parameters are listed is not important except for device descriptions, they must follow DEVICES=.

Most parameters should have default values. Exceptions are: LNAME, DEVICES, and device descriptions(ID and RATE). Consequently, most parameters are optional. Default values of these parameters are specified in the source file SIMULATOR.C.

EXAMPLE

```
LNAME=         OSSIM
DEVICES=       2
ID= PRNT  RATE=  10
ID= DISK  RATE= 300
TIME=          NSEC
TERMINALS=     4
MEMSIZE=       1000
CPURATE=       500
MAXSCRIPT=     2
MAXSEGMENTS=   1
SCHED=         FCFS
DEBUG_EVTQ=    OFF
DEBUG_MEM=     OFF
DEBUG_PCB=     OFF
DEBUG_RBLIST=  OFF
DEBUG_RBQ=     OFF
DEBUG_CPUQ=    OFF
```

===

LOGON.DAT: This file normally contains only LOGON events. During the first part of the project, this file will be used to enter events of all types. Each line of the file defines a single event. The format is the following:

```
Event   Agent_Id    Time
```

The "Event" field must be one of the following: LOGON, SIO, WIO, END, and EIO. Your program should be able to handle event names in upper or lower case, or a mix.

The "Agent_Id" field must designate a user or a device. User designations have the form "Uddd" or "uddd" where ddd is a 3-digit user number that must be <= TERMSIZE (the size of the terminal table). Otherwise, Agent_Id must be a valid device Id (again, insensitive to case).

The "Time" field must be an unsigned long decimal integer denoting a time relative to the beginning of simulation. The units associated with this time is determined by the CONFIG parameter TIME=. (Refer to the simulator function Convrt_time() in SIMULATOR.C when converting from external time units to internal "simtime".)

EXAMPLE

```
LOGON     U001       0
LOGON     U005      23
Logon     U002     400
LOGON     U003      13
SIO       disk     127
end       U003     100
```

==

SCRIPT.DAT: This file contains all process scripts need for the simulation. There must be one complete script for each LOGON event in LOGON.DAT. The order in which LOGON events occur during simulation determines the order scripts will be read from this file and the user process with which they are associated. For the sample LOGON.DAT file illustrated above, the first process script would be associated with U001, the second with U003, the third with U005, and the last with U002.

The format of the SCRIPT file is free form. Each script must be composed of a sequence of valid program names separated by blanks and ending with LOGOFF. The valid program names are: EDITOR, PRINTER, COMPILER, LINKER, USER.

To be fail safe, your program should accept these names in either upper or lower case (or a mix). IMPORTANT: as you read a script, you should convert the names to UPPER CASE; this is necessary for looking up these names in a table.

EXAMPLE

```
EDITOR
EDITOR  PRINTER
LINKER     LOGOFF  COMPILER  LOGOFF  PRINTER
user LinKer LOGOff

first script = EDITOR EDITOR PRINTER LINKER LOGOFF
Secnd script = COMPILER LOGOFF
third script = PRINTER USER LINKER LOGOFF
```

==

EDITOR.DAT
PRINTER.DAT
COMPILER.DAT
LINKER.DAT
USER.DAT

These files hold program scripts defining instances of each program type. For example, EDITOR.DAT holds all instances of type EDITOR. The number of complete program scripts in each file must be the same as the number of occurrences of the program name in SCRIPT.DAT. Using the example above, we see that the PRINTER program is executed twice, once in the first script, and once in the third script. Therefore, PRINTER.DAT must have two complete program scripts - and these scripts need not, and most frequently, will not be the same. Similarly, EDITOR.DAT and LINKER.DAT both have two scripts, while COMPILER.DAT abd USER.DAT each have just one script.

A program script must begin with information describing each segment of the program. This information is specified by two statements, the PROGRAM and SEGMENT statements. Their format is shown below.

```
PROGRAM  #segments
SEGMENT  length  access_byte
SEGMENT  length  access_byte
          ...
SEGMENT  length  access_byte
```

The PROGRAM statement must be first and specifies the total number of program segments. Following the PROGRAM statement is a series of SEGMENT statements, one for each segment. The SEGMENT statement specifies the length of the segment in no. of instructions, and the access byte. The access byte is given as hex literal (0xdd). In theory, the access byte can be used to specify the access restrictions associated with the segment (READ, WRITE, EXEC, APND).

Following the program header (PROGRAM and SEGMENT statements), the program script for each segment must be given. The length of each segment script must agree with the corresponding SEGMENT statement. IMPORTANT: each segment script must end with either a SKIP-JUMP pair or an END. The ONLY way control can be transferred from one segment to another is via a SKIP-JUMP pair!!!!!!

EXAMPLE

```
PROGRAM  2
SEGMENT  6  0x23
SEGMENT  5  0x2F

SIO      100
DISK     450
SIO       25
PRNT     100
SKIP       0
JUMP   [1,0]

WIO       31
REQ    [0,1]
WIO        0
REQ    [0,3]
END      678
```

================================

4.4 The Output File.

The contents of this file consists of the following information:

(1) An echo of the CONFIG parameters;
(2) A log of all events generated during simulation;
(3) An echo of each process script read (this should be produced immediately after the LOGON event that caused the script to

be read);

(4) The memory image of each program segment loaded; this should be produced following (3) or a program END event. Each segment should be printed so that the absolute memory location holding each instruction is displayed.

(5) Any debug output you may wish to produce.

When in doubt about what information should be produced and the format to be used, refer to the output (ossim.out) of the instructor's simulator (OSSIM).

EXAMPLE

```
+-----------------------------------------------------+
|  COP 4600, Summer '98              Simulator Report |
|  DATE:  Wed Aug 29 09:09:36 1998                    |
|  NAME:  Franke                                      |
|                                                     |
|  DELTA TIME ...........................NSEC         |
|  NO. OF TERMINALS .....................4            |
|  MEMORY SIZE ..........................1000         |
|  CPU RATE (NSEC/INSTR) ................500          |
|  SCHEDULING ALGORITHM .................FCFS         |
|  MAX SCRIPT LENGTH ....................2            |
|                                                     |
|  DEBUG MEMORY .........................OFF          |
|  DEBUG EVENT_Q ........................OFF          |
|  DEBUG PCB.............................OFF          |
|  DEBUG RBLIST..........................OFF          |
|  DEBUG RBQ.............................OFF          |
|  DEBUG CPUq ...........................OFF          |
|                                                     |
| ================  DEVICE TABLE ================== |
|                                                     |
|  ID      RATE(*)                                    |
|                                                     |
|  PRNT    10                                         |
|  DISK    300                                        |
|                                                     |
| (*)Bytes/Second                                     |
+-----------------------------------------------------+

EVENT  AGENT  HR:xxxxxxxx MN:xx SC:xx MS:xxx mS:xxx NS:xxx
-----  -----  --------------------------------------------

LOGON  U001          HR:0        MN:0  SC:0  MS:0   mS:0   NS:0

        SCRIPT FOR PROCESS U001 =
        EDITOR PRINTER

        SEGMENT #0 OF PROGRAM EDITOR OF PROCESS U001
        DISK ADDR: 28   LENGTH: 5
        MEM ADDR  OPCODE  OPERAND

              0    SIO     5
              1    PRNT    300
              2    CPU     0
              3    REQ     [ 0,  1]
              4    END     5

         PROGRAM EDITOR HAS BEEN LOADED FOR PROCESS U001

LOGON  U002          HR:0        MN:0  SC:0  MS:0   mS:0   NS:0

        SCRIPT FOR PROCESS U002 =
        PRINTER

        SEGMENT #0 OF PROGRAM PRINTER OF PROCESS U002
        DISK ADDR: 29   LENGTH: 11
        MEM ADDR  OPCODE  OPERAND

              5    SKIP    1
              6    JUMP    [ 0,  10]
              7    SKIP    1
              8    JUMP    [ 0,  10]
              9    SKIP    1
             10    JUMP    [ 0,  10]
             11    SKIP    1
             12    JUMP    [ 0,  10]
             13    SKIP    1
             14    JUMP    [ 0,  10]
             15    END     25

         PROGRAM PRINTER HAS BEEN LOADED FOR PROCESS U002

LOGON  U003          HR:0        MN:0  SC:0  MS:0   mS:0   NS:0

        SCRIPT FOR PROCESS U003 =
        EDITOR PRINTER

        SEGMENT #0 OF PROGRAM EDITOR OF PROCESS U003
        DISK ADDR: 125   LENGTH: 11
        MEM ADDR  OPCODE  OPERAND

             16    SIO     5
             17    PRNT    1000
             18    SKIP    0
```

```
               19    JUMP    [ 0, 6]
               20    CPU     0
               21    REQ     [ 0, 1]
               22    SKIP    1
               23    JUMP    [ 0, 10]
               24    SKIP    0
               25    JUMP    [ 0, 4]
               26    END     5

           PROGRAM EDITOR HAS BEEN LOADED FOR PROCESS U003

  SIO    U001          HR:0        MN:0  SC:5  MS:0   mS:0   NS:0
          START IO ON device PRNT
  WIO    U001          HR:0        MN:0  SC:5  MS:0   mS:0   NS:0
           PROCESS U001 IS BLOCKED FOR I/O.
           CPU BURST WAS 5 INSTRUCTIONS

  END    U002          HR:0        MN:0  SC:30 MS:0   mS:0   NS:0

           PROGRAM (PRINTER) ENDS ON TERMINAL= U002
           CPU BURST WAS 25 INSTRUCTIONS

  ==========>PROCESS FOR TERMINAL U002 HAS TERMINATED!


                              TERMINAL  SUMMARY
=========================================================================
TERMNL|       EXEC      |      WAIT     |    BLOCKED    |   ELAPSED     | EFF
======|================|==============|==============|==============|====
=========================================================================
TOTALS|          0 SC |          0 SC |          0 SC |          0 SC |
      |          0 NS |          0 NS |          0 NS |          0 NS |
------|----------------|--------------|--------------|--------------|
AVERGE|          0 SC |          0 SC |          0 SC |          0 SC |
      |          0 NS |          0 NS |          0 NS |          0 NS |
=====================================================================


                                DEVICE  SUMMARY
=========================================================================
DEVICE|       BUSY      |      WAIT     |      IDLE     |   RESPONSE    |%UTL
======|================|==============|==============|==============|====
CPU   |          0 SC |          0 SC |          0 SC |          0 SC |0.00
      |          0 NS |          0 NS |          0 NS |          0 NS |
=====================================================================
```