TO: MSPM Distribution
FROM: J. H. Saltzer
SUBJ: BL.0
DATE: 04/28/67

Multics system initialization has been completely redesigned
since October, 1966, when the previous overview was published.
Enclosed is a new overview, BL.0, which describes the
new design. The major improvement in the design is that
the file system paging mechanism is initialized very early,
so that it may be used during the remainder of initialization.

Identification

Overview of the Multics System Initialization
A. Bensoussan

Purpose

This section provides an overview of the various steps
taken in initializing Multics.

Introduction

The various initialization segments as well as the segments
that will comprise the hardcore supervisor of Multics
are all stored on a single magnetic tape known as the
Multics System Tape (MST).

From the time the bootload button is pushed to the time
Multics is initialized, 3 different stages are to be
distinguished, corresponding to 3 programs of different
nature:

1.   The bootload program
2.   The bootstrap initializer
3.   The Multics initializer

The bootload program is a special program in this sense
that it is located in the GIOC diodes and therefore it
is part of the 645 hardware. Furthermore, it is executed
in absolute mode.

The bootstrap initializer is a higher level program; although
it is entered in absolute mode, its first worry is to
set up a descriptor segment and to institute the appending
mode. However, it is still a "handicapped" program since
it does not tolerate any fault; as a consequence it cannot
use the standard call-save-return macros.

As for the Multics initializer, it can be coded like a
normal Multics program, using the call-save-return macros,
the dynamic linkage, the dynamic core allocation following
a missing page fault, and the dynamic descriptor segment
fabrication following the missing segment fault. The

linkage, missing page and missing segment fault handlers
have been made available by the bootstrap initializer;
they are not, of course, the standard Multics handlers
but they perform the functions expected by the Multics
initializer programmer. It is precisely one of the most
important Multics initializer's duties to make the Multics
handlers available for the above mentioned faults.

Having justified the existence of the 3 stages by the
different nature of the programs that comprise them, we
give below a brief description of each stage.

The bootload program

The GIOC contains, in fixed storage, 64 words which are
transferred into selected locations in core memory when
a button (known as the "bootload button") is pushed.
The 64 words are stored in the GIOC in a form of a diode
matrix and the base of the locations in core into which
they are transferred is determined by panel switches on
the system control console. As soon as the 64 words are
read into core, an interrupt is sent to the processor
which services interrupts on this GIOC. The interrupt
transfers the processor to the base of the block into
which the 64 words have been stored. This processor is
the only processor which will be executing during most
of the initialization sequence.

The diode matrix contains the "bootload program". The
bootload program is expecting a tape to be ready on the
tape handler connected to channel 31, device 0. If for
some reason, channel 31 is not available, any other channel
connected to a tape handler can be used; in this case,
the channel number has to be made known by the operator
to the bootload program through the 36 bit processor switches
as described in BC.4.01. Furthermore, the bootload program
is designed to read a tape written in the Multics standard
format (BB.3.01); briefly, any information recorded on
a Multics tape has in front of it a "label" that ends
with an end of file physical record.

When entered, the bootload program takes the following
actions:

1. It skips the label until the end of file
    physical record is encountered.

2. It reads into core the next physical record.

3. It transfers to the loaded program at a conventional
    entry point, passing interface information through
    index registers.

The bootload program is designed to read any Multics tape
provided that it satisfies the entry point convention;
this tape may contain the Multics system or any dump or
diagnostic program. For this reason, the bootload program
is not documented in BL sections (initialization) but
rather in BC sections (hardware).

In our case, the tape is the Multics system tape; therefore,
the bootload program loads the portion of the bootstrap
initializer contained in the first physical record following
the label and transfers to its entry point.

The bootstrap initializer

The bootstrap initializer is the interface between the
bootload program and the Multics initializer. As the
reader will notice, the Multics initializer is a very
large program which has an impressive number of functions
to perform. Therefore, "he" positively refuses to do
any work in the poor environment left behind the bootload
program. "He" wants "somebody" to load and initialize
a few of "his" segments; that is the sine qua non condition
for "him" to be able to work. "Somebody", of course,
is the bootstrap initializer. Therefore, the bootstrap
initializer starts reading the rest of itself, then it
reads and initializes the few segments mentioned above,
whereby providing the Multics initializer with:

1. An initialized descriptor segment

2. Paired base address registers, according to the
    Multics standard convention so that call-save-
    return macro can be used.

3. A loader, known as the "segment loader", that is
   capable, with the help of its utility routines,
   of loading segments from the MST

4. A fault handling mechanism such that

   a. When a linkage fault occurs, the fault can be
      replaced by the correct machine address, if the
      referenced segment is in core

   b. When a missing page fault occurs memory is
      dynamically allocated

   c. When a missing segment fault occurs, an appropriate
      segment descriptor word and the page table are
      manufactured.

The first instructions of bootstrap initializer execute
in absolute mode and they create a descriptor segment.
From this point on, all the stages of the initialization
can execute using the appending hardware.

For each segment that it reads from the MST, the bootstrap
initializer creates a segment descriptor word that is
appended to the descriptor segment, and an entry in a
segment known as the "Segment Loading Table" (SLT). In
this statement, three important objects have been mentioned:
the MST, the descriptor segment and the SLT; some comments
would be wise at this point.

The MST contains all the segments that need to be loaded
during Multics initialization: the initialization segments,
as well as the segments that comprise the hardcore supervisor,
as well as the segments that comprise the basic file system
hierarchy in case the hierarchy must be reloaded. Each
of these segments recorded on the MST, whether it is a
"text" or "link" segment, is preceded by an additional
piece of information: its "header". This header contains,
among other things, the name of the segment, its current
length, its maximum length, its access right, its status,
the page size, its type (initialization segment or supervisor
segment), etc. Briefly it contains anything that might
be needed at any point of the initialization.

The SLT is the segment in which all the headers recorded
on the MST will be saved during initialization. Each
time a segment is to be loaded from the MST, its header
is loaded first, since it precedes the segment; this header
is used to manufacture a fixed length entry for the segment
in the SLT.   Therefore, the SLT contains an entry for
each initialization or hardware supervisor segment which
are currently known during system initialization. Creating
an SLT entry for a segment implies assigning to it a segment
number because the SLT is indexable by segment number.
After the SLT entry is created, a segment descriptor word
has to be built in the descriptor segment.

The descriptor segment created by the bootstrap initializer
will be used throughout the rest of system initialization.
The structure chosen for the descriptor segment is based
on the following remark: There is no reason why hardcore
supervisor segments should have different segment numbers
during Multics initialization and during Multics operation.
Therefore, in the descriptor segment, segment numbers
0 to n are reserved for hardcore supervisor segments while
segment numbers greater than n are reserved for initialization
segments.   This structure gives the following advantages:
first, as far as the linkage mechanism is concerned, any
hardcore supervisor which has been "prelinked" can still
be used by the Multics initializer; second, when all the
hardcore supervisor segments have been loaded from the
MST, the upper part of the descriptor segment is the "template"
descriptor segment used at process creation time.

When the set of segments needed by the Multics initializer
to be able to stand alone have been loaded and initialized,
the bootstrap initializer calls the Multics initializer's
main program known as "Initializer Control Program", using
a standard call macro instruction, since it now is available.

The Multics initializer

It is in fact at this stage that the whole initialization
takes place.   The purpose of the two previous stages,
bootload program and bootstrap initializer, was merely
to provide the Multics initializer with the minimum machinery
needed for being able to run alone properly.

The Multics initializer is basically concerned with the
following functions: It makes known the hardware configuration,
initializes the secondary storage devices, loads and pre-links
the hardcore supervisor, initializes per-system data bases,
initializes per-process segments, and per-process entries
in the system tables in such a way that the Multics initializer
becomes a Multics process, and it creates the necessary
system processes so that Multics can stand alone. Then
it passes control to the Multics system control, that
is no more part of the Multics initializer.

1. Make known the hardware configuration. The hardware
    configuration is described in several segments that
    are recorded on the MST. Let us assume that K segments
    are needed. Call each of these segments CONF.SEG(k)
    where k=0,1,...K. The names of these segments are given
    in a loadlist, which is also a segment. Call this
    loadlist segment CONF.LL. At the time the Multics
    system tape is created, the desired configuration may
    not be known; therefore, the MST contains a description
    for a certain number I of possible configurations.
    For a given configuration CONF(i), the loadlist CONF.LL(i)
    contains the names of the K(i) segments needed to describe
    this configuration. Call each of the segments
    CONF.SEG(i,k) where k=1,2,...k(i). In the MST, the
    information is organized into "collections". A
    collection is a logical file. All the segments
    CONF.LL(i), for i=1,2,...I, are contained in the same
    collection, the "configuration loadlists" collection.
    All the segments CONF.SEG(i,k), for i=1,2,..,I and
    k=1,2,...K(i) are contained in the next collection, the
    "configuration library" collection.

    The Multics initializer is able to interact with the
    operator during initialization. It first asks the
    configuration number i; then it loads the corresponding
    configuration loadlist CONF.LL(i) and asks the operator
    if he wants to update it; the loadlist is changed
    according to the operator's requests and every segment
    whose name appears in the configuration loadlist is
    loaded from the configuration library collection. Then
    the operator has the ability of changing the content
    of any CONF.SEG(i,k) segment that has been loaded. When
    this is done, using the content of all the segments
    CONF.SEG(i,k) that have been loaded (and eventually
    updated), the Multics initializer builds the system
    configuration table.

2. Initialize secondary storage. If the file system
    hierarchy must be reloaded, free storage maps are
    written on all secondary storage devices available
    to the file system. Then, areas of secondary storage
    to be used by this version of Multics are defined, the
    root directory is initialized and the segments needed
    to operate the hierarchy reconstruction process are
    loaded in secondary storage.

3. Load the hardcore supervisor. Like the configuration
    segments, the hardcore supervisor segments are
    mentioned in a loadlist. The MST may contain a
    certain number J of hardcore supervisor versions.
    For a given supervisor version SUP(j), the loadlist
    SUP.LL(j) contains the names of the K(j) segments
    that make up this supervisor version. Call each
    of these segments SUP.SEG(j,k) where k=1,2,...,K(j).
    In the MST, all the segments SUP.LL(j), for j=1,2,...,J,
    are in the "supervisor loadlists" collection. All
    the segments SUP.SEG(j,k), for j=1,2,...,J and
    k=1,2,...,K(j) are contained in the "supervisor
    library" collection.

    The operator is asked for the number j of the supervisor
    version he wants. The Multics initializer loads the
    corresponding supervisor loadlist SUP.LL(j). Then
    the operator is asked if he wants to update it; the
    loadlist is changed according to his requests. Then
    every segment SUP.SEG(j,k) whose name appears in the
    SUP.LL(j) segment (eventually updated) is loaded
    from the supervisor collection library. The operator
    does not have the ability of requesting to change
    any of the segments loaded from the library collection.

4. Prelink the hardcore supervisor. In the hardcore ring
    of the Multics supervisor, dynamic linking is not a
    necessary feature, since all modules to be used in
    any single version of the hardcore supervisor are known
    at initialization time. Thus, a dynamic link and
    search mechanism will accomplish nothing that pre-linking
    at initialization time cannot do. Since in addition
    the majority of all processes share the same copies
    of virtually all data and procedure segments of the
    hard-core ring, pre-linking these data and procedure
    segments permits the linkage sections themselves to
    be shared also. Prelinking therefore has two purposes:

1) The Multics dynamic linking and search mechanism does not have to be a part of the hardcore ring, and does not have to be "wired down", and

2) The number of per/process segments is reduced.

5. Initialize per-system data bases. They are:

a. The file system data bases: the core map, the hardcore segment table (HST), the device disposition table (DDT), the data bases (buffers, DIM history) of each device interface module (DIM), the active segment table (AST), the descriptor segment table (DST), the process segment table (PST), the process waiting table (PWT).

b. The traffic controller data bases: the known process table (KPT), the active process table (APT), a processor data segment for each processor, the template descriptor segment, stack history.

c. The fault-interrupt interceptors data bases: the fault interrupt vector, the vector redirector segment, the system communication segment, the processor communication segment.

d. The I/O system data bases: GIOC interface module (GIM) data bases, tape controller interface module (TCIM) data bases.

Although it is not an exhaustive list of the system data bases, this list contains the major system data bases that have to be initialized.

6. Initialize per process data bases. A Multics process is characterized by 5 basic segments that are created in the process creation module. These segments are: process directory, known segment table, process data segment, process definition segment and hardcore stack. They must be created and initialized on the behalf of the Multics initializer.

7. Create per-process entries in system data bases. An
    entry for the Multics initializer process is created
    in the known process table (KPT), in the active process
    table (APT) and in the process segment table (PST).
    One entry is created in the descriptor segment table
    (DST) for the descriptor segment of the Multics
    initializer. An active file trailer (AFT) is associated
    with every AST entry, showing that the corresponding
    segment is active on the behalf of the Multics
    initializer.

8. Create one branch for each existing segment in the
    appropriate subtree of the hierarchy.

9. Create necessary system processes. When the functions
    described above have been done, the Multics initializer
    presents all the characteristics of a Multics process,
    with the only restriction that it is the only process
    in the system and therefore it cannot afford to be
    blocked. In order to negate this restriction, necessary
    system processes are created, using the standard process
    creation mechanism. The system processes to be created
    by the Multics initializer are specified in the system
    configuration table. They can be for instance the
    file system device monitor process, an idle process
    assigned to a processor, etc.

At this point the Multics initializer is a full active
and loaded Multics process. The work assigned to it is
completed and the Multics initializer issues a call to
the system control procedure:

        call <Multics> | [system_control].

When this call is given, the Multics initializer evolves
into the System Control Process, and Multics is now "in
operation". The system control process creates the other
members of the system control process-group; then it creates
the answering service process to allow Multics users to
"dial-up" the computer.

Note that the system control is not part of the Multics
initializer. It is documented in BQ section.

After having read this paragraph, the Multics initializer
might appear to be a straightforward program, with 9
independent modules to perform the 9 functions mentioned.
The main disadvantage of this approach is that it would
require a large amount of special purpose code to perform
functions that can be done automatically by the hardcore
supervisor. Therefore the strategy that has been chosen,
which is described in BL.5.00, is basically as follows:
A group of hardcore supervisor segments is loaded and
initialized; then another group of hardcore supervisor
segments is loaded, and it is initialized using the hardcore
supervisor segments made available previously and so on.

The second disadvantage of the straightforward method
is that it would require to have in core, at the same
time, all the hardcore supervisor segments even if they
do not need to be wired down when Multics is operating.
For the present time, the size of the hardcore supervisor
is such that this requirement cannot be satisfied. In
the strategy that has been taken, only the wired-down
segments of the hardcore supervisor need to be in core
at the same time during Multics initialization.

Identification of a Multics System

When a Multics System is initialized it is identified by
three items:

1. Tape identification
2. Configuration identification
3. Supervisor identification

The tape identification (TI) is a unique name recorded
on the Multics system tape when it is created, for example
"MAC23"

The configuration identification (CI) is the name of the
configuration loadlist selected by the operator, for
example, "18"

The supervisor identification (SI) is the name of the
supervisor loadlist selected by the operator, for example
"L"

A typical system identification might then be MAC23.18.L

If operator editing of the configuration list or load list
has occurred, a "single quote" (prime) character is
appended to the appropriate part of the system identification,
e.g., MAC23.18'.L

Organization of BL sections

As it can be noticed, the first stage "bootload program"
is not part of BL sections. It is documented as part
of the hardware, in BC.4.01. All the data bases used
by the "bootstrap initializer" and by the Multics initialize
itself: the MST, the SLT and the system configuration
table are grouped at the beginning of the BL chapter under
BL.1,2 and 3. BL.4 is the bootstrap initializer. BL.5
is the Multics initializer. The point this paragraph
wants to make is that sections BL.6,7,8,9,10 and 11 have
to be thought of as BL.5 subsections since the modules
they describe: segment loader, initialization linker,
I/O initializer, fault-interrupt initializer, file system
initializer and traffic controller initializer, are part
of the Multics initializer.

```
   ( Bootload  )              (  TAPE  )                    (  TAPE  )      ( OPERATOR )
   (  Button   )                  |                            |               |
        |                         |                            |               |
        |                         |                            |               |
  ┌─────────────────┐    ┌──────────────────────────┐   ┌──────────────────────────────────┐   ┌──────────┐
  │                 │    │                          │   │                                  │   │          │
  │ BOOTLOAD        │    │      BOOTSTRAP           │   │          MULTICS                 │   │ SYSTEM   │
  │ PROGRAM         │───▷│      INITIALIZER         │──▷│          INITIALIZER             │──▷│ CONTROL  │
  │                 │    │                          │   │                                  │   │          │
  ├─────────────────┤    ├──────────────────────────┤   ├──────────────────────────────────┤   └──────────┘
  │ Skip MST Label  │    │ Set appending mode       │   │ Make known configuration         │
  ├─────────────────┤    ├──────────────────────────┤   ├──────────────────────────────────┤
  │ Read BI first   │    │ Read necessary MI segments│  │ Initialize secondary storage     │
  │ record          │    ├──────────────────────────┤   ├──────────────────────────────────┤
  ├─────────────────┤    │ Initialize SLT           │   │ Load hardcore supervisor         │
  │ Transfer to BI  │    ├──────────────────────────┤   ├──────────────────────────────────┤
  └─────────────────┘    │ Initialize linkage fault │   │ Pre-link hardcore supervisor     │
                         │ handling                 │   ├──────────────────────────────────┤
                         ├──────────────────────────┤   │ Initialize per system data bases │
                         │ Call MULTICS INITIALIZER │   ├──────────────────────────────────┤
                         └──────────────────────────┘   │ Initialize per process data bases│
                                                        ├──────────────────────────────────┤
                                                        │ Initialize per process entries in│
                                                        │   per system data bases          │
                                                        ├──────────────────────────────────┤
                                                        │ Create branches in Hierarchy     │
                                                        ├──────────────────────────────────┤
                                                        │ Create necessary system processes│
                                                        ├──────────────────────────────────┤
                                                        │ Call SYSTEM CONTROL              │
                                                        └──────────────────────────────────┘
```