TO: Multics Distribution
FROM: R. M. Graham
SUBJECT: BH.4.02
DATE: August 17, 1967

Change published date - 1/04/66 to 1/04/67.

Identification

The Reload I/O Process
S. H. Webber

Purpose

This section describes the organization and design of
the reloading I/O process used by the hierarchy reconstruction
process and the secondary storage reload process (BH.3).

Introduction

The reload I/O process handles the I/O for all reload
processes. Communication between the I/O process and
the reload processes is done through an I/O queue. The
various reload processes approach the queue at different
(random) times. The I/O process must merely keep up with
work assigned to it by the reload processes. The backup
scheme includes multiple reload processes to insure as
fast a reload as possible given the several processors
of a typical Multics installation.

The interaction between the I/O process and the reload
processes is depicted in figure 1. The several reload
processes make calls to the several entry points to the
call handler routine when appropriate. The call handler
then creates a new queue entry in the I/O queue, stores
into this entry the process identification and reloader_id
(passed as an argument in the call) and reconnects the
various pointer chains which define the queue.

Associated with each reload process is a working directory
for that process. In this working directory are several
buffers and data bases through which any data transferred
to the I/O process actually flow. The I/O queue entry
merely specifies which reloader (and consequently which
buffers) are responsible for the I/O request. There are
two variables which are placed in each queue entry. These
are:

1)   The process identification of the reload process
      making the call. This is used to wakeup the reload
      process at a later time, and

2)   The reloader identification. The "reloader_id" is
      a number used to identify the reload process to the
      I/O process. The numerical value of the "reloader_id"
      is used by the I/O process as an index into several
      of its tables.

The directory name, of course, allows unambiguous access
to the correct buffers to be associated with the request.
All reload processes are identical and hence corresponding
buffer segments (directly inferior to the reload working
directory) for different reload processes all have the
same name. These latter names are known by the I/O process,
and hence the only additional information needed in any
one request is the directory name. (The directory path
name is supplied at initialization time.)

On the other side of the queue, the I/O process goes blocked
awaiting a wakeup signal from some reloader. When it
finds a new request upon awakening it selects an input
unit, copies the queue entry, deletes the entry in the
queue, reads in the header from tape, stores the header
information in the appropriate buffer, sets the status
switches OFF, and starts I/O on the preamble and data
segments. (This will be discussed in more detail later.)

When a reloader is initialized it calls the I/O process
at the following initialize entry point:

        call init_reloader (path_name, reloader_id);

where

        dcl   path_name char(*),
                reloader_id bit (36);

applies.

path_name is the directory path name of the working directory
for this reload process. The I/O process then makes the
buffer segments of this reload process known to it and
stores the segment numbers in a table which it keeps for
its own reference. This table is indexed by the numerical
value of "reloader_id". The PL/I declaration for this
table is as follows:

        dcl   1   table (max_proc),

                2   header_buffer ptr,

                2   preamble_buffer ptr,

                2   directory_buffer ptr;

If chfx is a routine which converts the directory name
into its numerical value then a typical reference to the
header buffer of reload process proc 1 would be

        table(chfx(proc_1)).header_buffer -> ...

The PL/I declaration for the header buffer is as follows:

```
dcl  1  header ctl(hp),

     2  pre_status bit(1),

     2  dir_status bit(1),

     2  trap_sw bit(1),

     2  tmtype bit (17),

     2  uid bit (70),

     2  dtm bit (72),

     2  table_name char(12),

     2  table_index char(6),

     2  n fixed bin (17),

     2  slotno fixed bin (17),

     2  current_ln fixed bin(17),

     2  slot_name_ln fixed bin(17),

     2  slot_name char(hp ->header.slot_name_ln);
```

The next four sections describe respectively the reload
I/O queue, the reload I/O call_handler, the reload I/O
process, and the reload list (a list of reel labels used
to define the reload sequence).

The Reload I/O Queue

The reload I/O queue is a doubly threaded list of entries.
The entries are generated by the routine call_handler
common to all reload processes. With each call to getheader
(an entry point withing the call_handler routine) a new
entry is created in the queue by allocating storage for
it and setting the forward and backward pointer chains.
After the entry is so established, the process identification
and reloader_identification are stored in the queue and
the I/O process is sent a wakeup signal.

Simultaneously, the I/O process is blocked awaiting a
wakeup signal from a reload process. For each entry it

finds upon awakening it starts the I/O activity requested
and deletes and frees the queue entry. The header buffer
(directly inferior to the reloader's working directory)
is then loaded with the header information. The status
switches (explained later) are turned OFF and stored in
the header buffer.

The PL/I declarations for the queue (header) and the queue
entries are as follows:

```
dcl  1  ioq$header ext,

     2  lock bit (36),

     2  first_ptr bit (18),

     2  last_ptr bit (18),

     2  length fixed bin(17),

     2  var area (segment_size);

dcl  1  entry ctl(qp),

     2  forward_ptr bit (18),

     2  backward_ptr bit (18),

     2  reloader_id bit (36),

     2  process_id bit(36);
```

The Reload I/O Call Handler

The routine call handler is common to all reload processes
and has the following entry points:

    getheader

    getpreamble

    getdirectory

    getsegment

When getheader is called the call handler creates a new
entry in the queue. Into this entry it places the "reloader

identification" (found as an argument to the call) and the
"processid". The I/O process is then awakened to insure
that the newly created entry is processed and the reload
process then goes blocked. Upon awakening, the entire
header information will have been placed in the header
buffer for this reload process. The header buffer contents
are then merely copied into the argument list and the
call handler returns.

When either getpreamble, getdirectory, or getsegment is
called, the call handler assumes that a previous call
to getheader has been made. The call to getheader instructs
the I/O process to begin I/O on the preamble and, if present,
the directory or data segment. When the I/O process finishes
reading in the preamble (into the preamble segment for
the appropriate reload process) it turns the preamble
status switch (found in the header buffer for the process)
ON to so indicate. A similar action is taken upon com-
pletion of the I/O for the optional directory or data
segment.

When a call to getpreamble occurs, the call handler tests
the preamble status switch and if it indicates that the
preamble has been successfully read in, i.e. the switch
is ON, it returns. If the switch is OFF the call handler
calls block awaiting for a wakeup from the I/O process.
(Whenever the I/O process sets any status switch a wakeup
signal is sent to the appropriate reloader).

Calls to getdirectory and getsegment work exactly as do
calls to getpreamble in relation to status switch operations.

If a call to getpreamble, etc. occurs not preceded by
a call to getheader an error return is taken.

The call_handler is called in any of the following ways:

call getheader(reloader_id,srs,sfs,uid,dtm,incsw,dirsw,table_
name,table_index,n,slotno,slotname,errtn);

call getpreamble(reloader_no, errtn);

call getdirectory(reloader_no, errtn);

call getsegment(reloader_no, errtn);

Where the arguments are defined as follows:

reloader_id bit (36)   An identifier unique to each reload process. Each reloader_id has a different set of buffers (1 preamble, 1 header, 1 directory, etc.) associated with it.

srs bit(1)   segment-removed-switch (see BH.3)

sfs bit(1)   segment-follows-switch (see BH.3).

uid bit (70)   unique id of the terminal entry in the preamble

dtm bit (72)   date/time-file-last-modified of the terminal entry

incsw bit (1)   indicates whether the current record is considered to be incremental or checkpoint.

dirsw bit (1)   directory switch (ON if terminal entry is a directory branch)

table_name char(12)   the segment name of a table which contains the retrieval argument (reel label and record number) for the record currently being processed

table_index char(6)   An index into the above table.

n fixed bin(17)   number of entries in the preamble

slotno fixed bin(17)   slot number of the terminal entry

slotname char(*)   slotname of the terminal entry

errtn label   error return

Associated with each reloader_id (in addition to the buffers) is a "device scanning list" which gives the identification of the particular device wanted by the reloader. Ordinarily only the first entry in the list is non-random and is set to the device last used by the reloader associated with "reloader_id". When the I/O process then reaches for a device it tries to hook up to the device specified

by the first entry in the device scanning list. If the I/O
process is successful in procuring this device then maximum
use can be made of redundant preamble information. (Subse-
quent logical records on tape although completely independent
often define similar positions in the file system hierarchy
and therefore much of the preamble will be identical.
To take advantage of this redundancy, it is best to put
all such similar preambles and their data segment on the
same tape at dump time. This will allow the reloader
to take maximum advantage of the dumping scheme by using
the same device for subsequent calls).

The Reload I/O Process

The reload I/O process must perform the following tasks:

1)   Accept requests (from the I/O queue) for input of
     logical records from tape

2)   Keep track of the logical record hierarchy (subrecords,
     etc.)

3)   Handle end-of-tape error returns

4)   Specify which tape reels should be used for a reload
     and in which order they should be used

5)   Keep track of which buffers are associated with which
     reloader_id's and

6)   Keep track of the device scanning list for each reloader.

The manner of accepting requests (by searching the I/O
queue for new unprocessed entries upon response to a wakeup
signal) is straight forward. Each time the I/O process
completes its work for one reloader it searches the queue
again and goes blocked if there are no new entries. The
queue is loaded from the top and unloaded from the bottom
with the assumption that the reloaders cannot get too
far ahead of the I/O process. (The queue is set to a maximum
size above which the reloaders cannot extend it.)

The logical record hierarchy used by backup is described
in detail in BH.4.03 The general construction can be
described by the following outline (the hierarchical
character is then obvious):

1. Tape header

2. Tape records

    a) Record no. 1

      i. Header

      ii. Preamble

      (iii. Directory or data segment) - optional

    b) Record no. 2

      i. Header

      ii. Preamble

      (iii. Directory or data segment) - optional

      .

      .

      .

    c) Record no. n

      i. Header

      ii. Preamble

      (iii. Directory or data segment) - optional

When a call to _getheader_ is made the I/O process must find an available device, forward the tape to the beginning of the next logical reocrd, if not already there, and begin reading in the header data. Only after a complete record has been processed is the device made available for further calls to _getheader_.

End-of-tape returns are built in as a Multics I/O system facility. Upon detection of such a condition the I/O process must do the following:

  1) Instruct the I/O System to unload this tape reel,

  2) Search in the reload list (described later) for the next tape reel to be used, and

  3) Instruct the I/O system to load this reel (on the same device presumably).

The manner in which the reload list is used is described
in the next section.

The Backup I/O Reload List

As described in section BH.4.03 each tape created by the
various backup processes includes a header (created by
the Backup I/O process) which contains among other items
a list of the tape reels to be used at reload time (here-
after called the reload list). Whenever a secondary
storage reload is initiated, the last backup tape created
is loaded to initialize the reload process. The header
of this tape is then stored in a data base used by the
I/O process (as the reload list). As the reloader finishes
with a tape reel it instructs the I/O System to mount
a new reel and specifies the appropriate reel via its
reel label. Reels are reloaded in reverse chronological
order -`the last created is the first used.

While writing either type of checkpoint tape, the header
will not include that particular checkpoint dump in the
reload list. Checkpoint dumps do not exist until complete.
Incremental tapes, however, do include their own label
in the reload list. With this design the last tape created
(possibly the tape being created) contains the correct
reload list.

The I/O process determines which reel should be mounted
by the information found in the reload list (information
from reels already processed). Each header contains
complete information specifying which reels should be
used for the hierarchy reconstruction process. Each header
specifies reels through n user checkpoint dumps (n being
the user checkpoint retention period). Whenever a user
checkpoint reel is encountered the reload list kept by
the I/O process is reset to that in the header of the
newly found user checkpoint reel. Then, if the reloading
of the user checkpoint dump should not succeed completely,
a further (older) user checkpoint dump will have been
specified.

The hierarchy reconstruction process continues until a
complete system checkpoint dump has been reloaded.
When this occurs the second phase of the reloading process
is entered (the secondary storage reload process, BH.3.02).
This process continues to function until a complete user
checkpoint dump has been reloaded.

If a system checkpoint reel cannot be found or does not
exist the reloader skips this reel and notes that this
system checkpoint dump cannot be reloaded completely.
Those tapes that can be reloaded are, but the reloader
does not enter the secondary reload phase until a complete
system checkpoint dump has been reloaded. Once a complete
system checkpoint dump has been reloaded all further such
dumps are skipped. Note there are several system checkpoint
dumps for each user checkpoint dump and if none of the
former can produce a successful, complete reload by the
time a user checkpoint dump is encountered, the user checkpoint
must be skipped over. Each time the hierarchy reconstruction
process skips over such a user checkpoint dump, the reload
list is "updated" to include another (older) user checkpoint
dump.

In that the reload list reflects which backup tapes have
been created (in reverse order) the configuration is as
follows: the first reels specified are incremental tape
reels (each reel is a unit in itself). Then follows a
system checkpoint dump which may consist of several reels.
This entire sequence is then repeated several times before
a user checkpoint dump (of possibly several reels) is
encountered. This whole configuration is repeated once.
The following might be the layout of a typical (reel header)
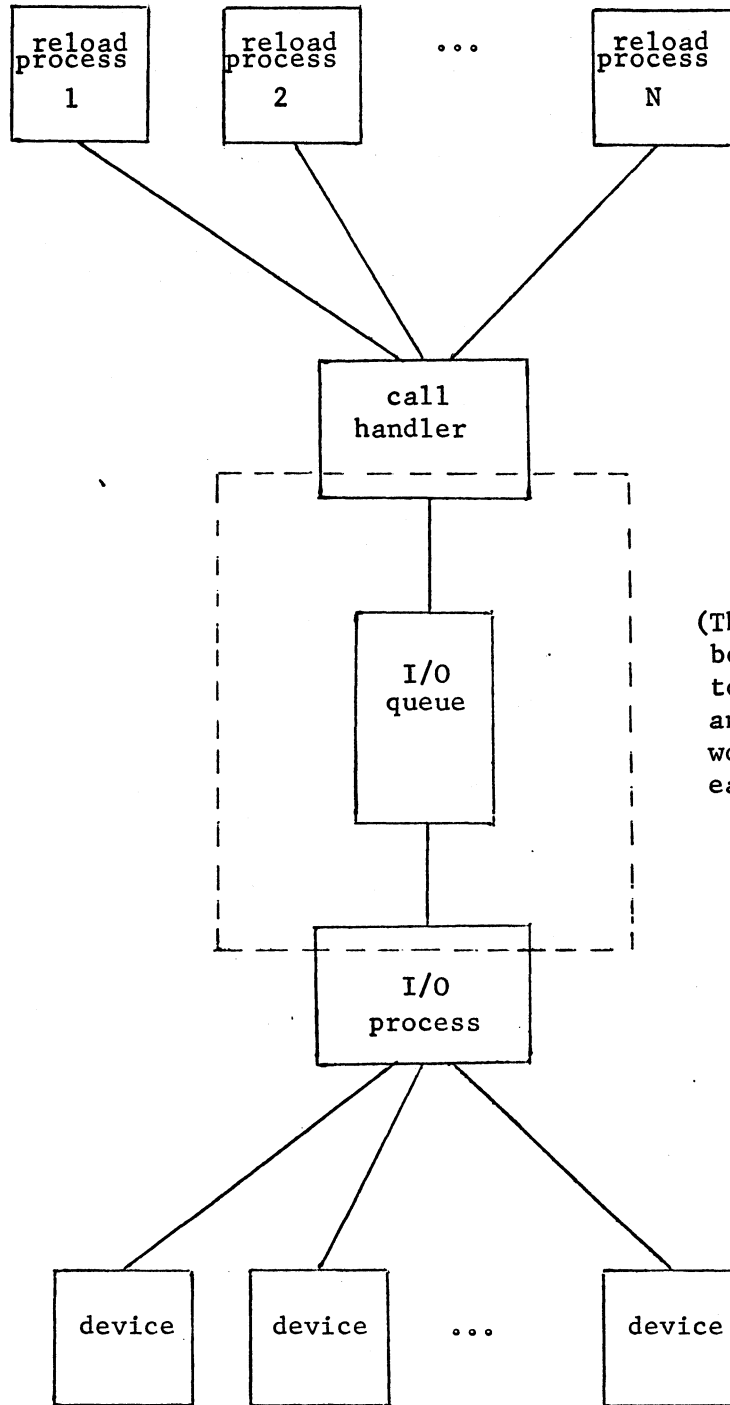reload list:

load first   load last
                                                (if necessary)

IIIISIIIISIIIISUIIIIISIIIIIISIIIISU

I represents 1 incremental tape

S represents 1 system checkpoint dump (possibly several tapes)

U represents 1 user checkpoint dump (possibly several tapes)

(The I/O queue can easily
be removed, as indicated,
to allow one dumping process
and one device process to
work synchronously on an
early version of Multics.)

Figure 1