Published: 01/06/67

Identification

The Dumping I/O Process
S. H. Webber

Purpose

The dumping I/O process is a utility process incorporated into the backup system to 1) insure minimum waiting times caused by I/O activity and 2) handle the I/O for several dumping processes simultaneously (BH.2.00).

The backup scheme includes multiple dumping processes to insure that with Multics expansion and, under conditions of extreme system load, adequate facilities are available for the dumping function. With several dumping processes working simultaneously (distributing their work over several device processes) there will be a faster, more uniform scan of the system hierarchy which will allow for a more consistent set of backup tapes.

This section describes the general configuration of this I/O process relative to the dumping processes and includes a detailed description of the I/O queue used therein.

Introduction

The dumping I/O process handles the I/O for all dumping processes. Communication between the I/O process and the dumping processes is done through an I/O queue. The I/O queue is loaded by the routine <u>ioroutine</u> (common to all dumping processes) and unloaded by the I/O process itself. Communication between any one dumping process and the I/O process is synchronous in that the dumping process cannot proceed until the I/O process has examined and begun to process the corresponding queue entry. On the other hand, the I/O process appears to work asynchronously with respect to the I/O system, for it approaches the queue for a new entry to process after it has started the I/O for the previous entry of the queue. The various dumping processes load the queue at random times - the I/O process must merely keep up with work given to it by the various dumping processes. (Note: The I/O process cannot get ahead of the dumping processes due to the synchronous behavior of the interaction.)

The modular description of the interaction between the I/O process and the dumping processes is depicted in figure

1. The several dumping processes make calls to ioroutine whenever appropriate.

On the other side of the queue the I/O process is blocked awaiting a wakeup signal from a dumping process. Upon awakening and finding an unprocessed entry it selects an appropriate device and initiates the I/O on that device. Several devices may be allocated to the I/O process; it must find and use those that are presently available.

Associated with each dumping process is a working directory for that process. In this directory are several buffers through which any data transferred to the I/O process actually flow. The I/O queue entry merely specifies which dumper (and consequently which buffers) is responsible for the I/O request.

There are 2 variables which are placed in each queue entry. These are

1. The process identification of the dumping process making the request. This is used to wakeup the dumping process at a later time. And

2. The dumper identification. The "dumper_id" is a unique identifier of the dumping process. The numerical equivalent of the "dumper_id" is used by the I/O process as an index into several of its internal tables.

   The directory name, of course, allows unambiguous access to the correct buffers to be associated with the request. All dumping processes are identical and hence corresponding buffer segments (immediately inferior to the dumper working directory) for different dumping processes all have the same name. These latter names (not path names) are known by the I/O process, and hence the only additional information needed in any one request is the directory name. (This path name is supplied at initialization time.) The path name of a buffer is formed by concatinating the dumping process directory path name with the buffer name.

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

```
call init_dumper(path_name, dumper_id);
```

where

```
dcl path_name char (*),
    dumper_id bit (36);      applies.
```

path_name is the directory path name of the working directory for this dumping process. The I/O process then makes the buffer segments of this dumping 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 "dumper_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 (character) into its numerical value (fixed) then a typical reference to the header buffer of dumping 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 slot_no 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 order in which backup tapes are referenced at reload time is specified in the "reload list", a list of reel labels located in the tape header of each backup reel created. This "reload list" is described in BH.4.02.

The next 3 sections will treat separately each of the following subjects: the I/O queue, ioroutine (the queue loading procedure), and the dumping I/O process.

The Dumping I/O Queue

The dumping I/O queue is a doubly threaded list of entries; that is, each allocation or entry in the queue must contain a forward and backward pointer. The set of forward (backward)

pointers within the queue then defines a continuous chain through the queue. There is, in addition, a queue header which contains a lock for, pointers to the head and tail of, and the current size of (i.e. number of entries in) the queue.

The queue is loaded by ioroutine and unloaded by the I/O process. The process which currently uses ioroutine goes blocked for the time after it loads the queue and before the I/O process can return the retrieval arguments.

The lock used will be a 36 bit bit string set by the standard interlock mechanism (the STAC instruction). The process attempting to lock is blocked if the queue is already locked.

The first and last pointers for the queue must be self-relative pointers each pointing to (probably) separate queue entries. If these pointers are zero then the list is empty. If they are equal, then the list has one entry in it.

The current size of the queue is an integer which specifies how many separate entries exist in the queue. This is used in scanning the queue for unprocessed entries and as a quick check to see if there are indeed any entries in the queue.

The PL/I declaration for the queue header is as follows: (var is an area in which the queue entries actually reside.)

```
dcl 1 ioq$header ext,
      2 lock bit (36),
      2 first_ptr bit (18),
      2 last_ptr bit (18),
      2 length fixed,
      2 var area (segment_size);
```

Each entry of the queue contains the following variables:

1. forward ptr — a pointer to the next queue entry

2. backward ptr — a pointer to the preceding queue entry.

3. dumper_id — the directory path name of the dumping process (acts as an index into various I/O process tables when converted to its numerical value).

4. The process_id of the dumping process (to be used by the I/O process to awaken the dumping process when the return arguments have been loaded in the queue).

The PL/I declaration for each queue entry is as follows:

```
dcl 1 entry ctl (qp),
    2 forward_ptr bit (18),
    2 backward_ptr bit (18),
    2 dumper_id bit (36),
    2 processid bit (36);
```

The preamble string (described fully in BH.4.03) is stored in the preamble buffer segment known to the particular dumping process and simultaneously to the I/O process. Although the segment numbers each process associates with this segment may be different, the "dumper_id" guarantees no ambiguity.

Each dumping process has, in addition to the preamble buffer segment and the header segment, another segment which contains (if called for) the directory segment pointed to by the terminal entry in the preamble. This buffer segment is handled in the same way the preamble buffer segment is and identification is again by means of the "dumper_id" variable.

The I/O Queue Loader

The I/O queue is loaded by the dumping routine ioroutine which is common to all dumping processes. This routine transfers data given to it as arguments into the header buffer. Some of this data points indirectly to data segments; the rest is specific information needed by the various reloaders. (See BH.3.01 and BH.3.02).

The call to ioroutine is given by:

```
call ioroutine (dumper_id,tmtype,m,uid,dtm,name,slot,length,
               table_name,table_index,errtn);
```

where:

dumper_id bit (36)	is the identifier unique to each dumping process. (i.e. the directory path name.)
tmtype bit (17)	is the terminal entry type.
m fixed bin (17)	is the number of entries in the preamble.
uid bit (70)	is the unique identification of the terminal entry.

dtm bit (72)	is the data/time the segment pointed to by the terminal entry was last modified.
name char (*),	is the slot name of the current directory being scanned by the dumper.
slot char (8),	is the slot number of the entry being processed by the dumper.
length fixed bin (17),	is the current length (in 64 word pages) of the file pointed to by the terminal entry.
table_name char (17),	is the name of a segment in which the retrieval arguments are stored.
table_index char (6),	is an index into the table "table_name" specifying which retrieval arguments to associate with the present argument list.
errtn label,	is an error return.

The work done by <u>ioroutine</u> is described below:

1. Check the queue status, if unlocked go to 3.
2. Since the queue is locked, call block, upon awakening go to 1.
3. Lock the queue.
4. If queue is too full for a new entry, go to 1 (after unlocking the queue and waking up any waiting processes).
5. Allocate area for a new queue entry.
6. Fill in the queue - (thread at end of list).
7. Unlock the queue (awaken processes waiting for the queue).
8. Wakeup the I/O process.
9. Call block.
10. Get table_name and table_index from the header buffer.
11. Return.

ioroutine allocates storage for the queue entries whereas the I/O process frees the storage of each queue entry.

While the dumping process is blocked, the I/O process finds an appropriate waiting device and determines and loads the table_name and table_index into the header buffer.

Figure 2 is a flow diagram for the procedure ioroutine.

The Dumping I/O Process

The dumping I/O process works in series with ioroutine in the sense that 1) the I/O process must wait for ioroutine to load the queue before it can take any action and 2) ioroutine cannot continue until the I/O process has performed certain processing of the appropriate entry.

Due to the scanning algorithm used by the dumping process subsequent calls from any one dumping process will request dumping of entries and segments whose preambles are quite similar. It is possible to take advantage of this duplication by having each dumping process associated with a specific device. This would allow the reload process (by using just one device and hence the results of 1 dumper) to decode only that part of the preamble which is not duplicated on subsequent logical records. This is not always possible but to insure that the attempt is made a "device scanning list" exists which associates each dumping process with several devices. Ordinarily the first entry of the list (associated with the device of primary interest) will be the only non-random one. However if it becomes necessary to change over to a second device for some reason, subsequent entries in the list specify available devices.

The work done by the I/O process is described below:

0. Block - wait for wakeup.

1. Check the queue status, if unlocked go to 3.

2. Since the queue is locked, call block, upon awakening go to 1.

3. Lock the queue.

4. If the queue is empty go to 0 (after unlocking the queue and waking up any waiting processes).

5. Free the current queue entry - reconnect the pointer chains

6. Unlock the queue (wakeup any waiting processes).

7. Store the appropriate table_name and table_index in the header buffer for the appropriate dumping process.

8. Wakeup the dumping process (the processid was in the queue entry).

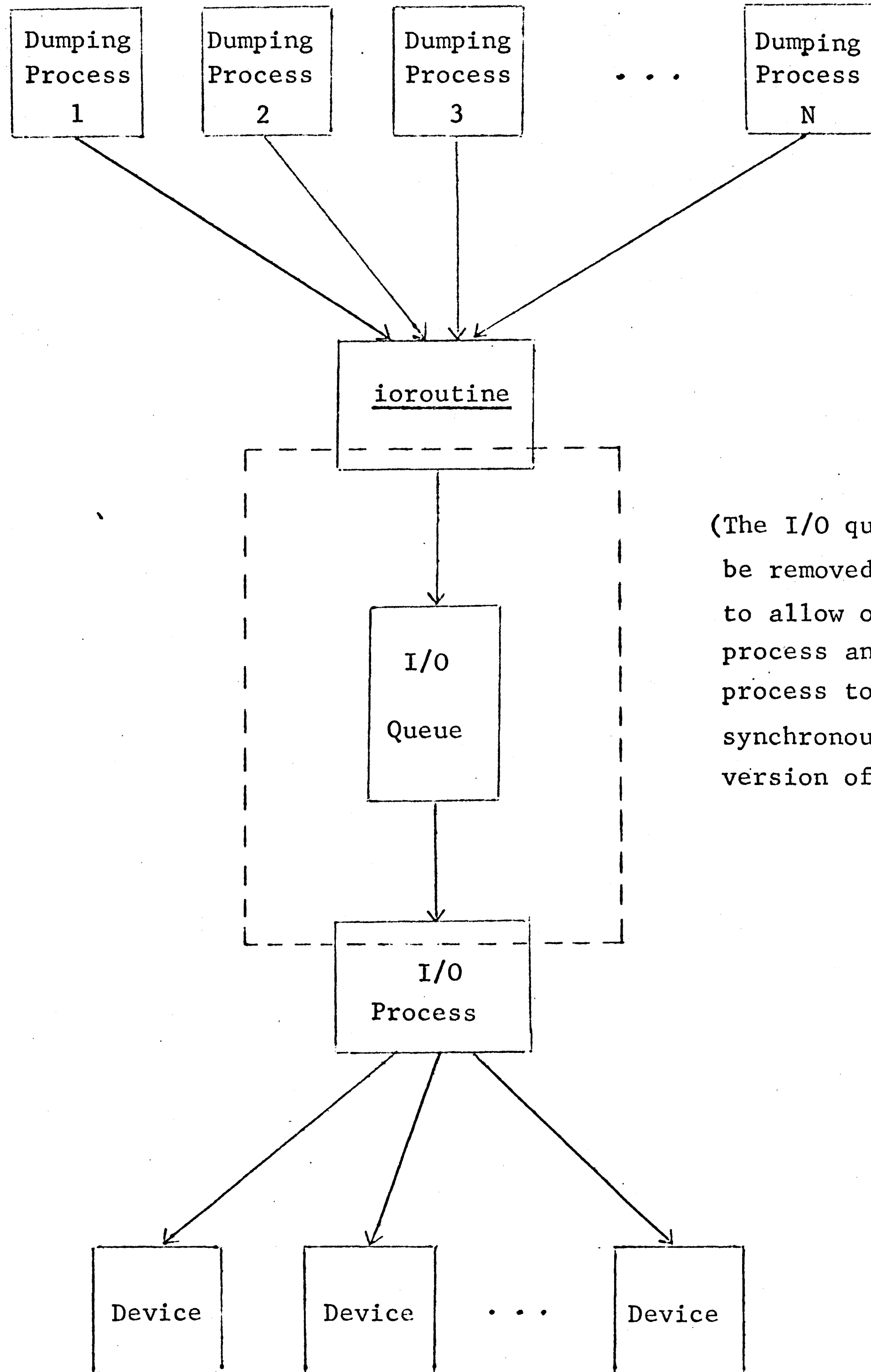
9. Select a waiting device (refer to the "device scanning list").

10. Start I/O on the selected device.

11. Make the map entry (copy the preamble segment into the MAP segment).

12. Go to 0.

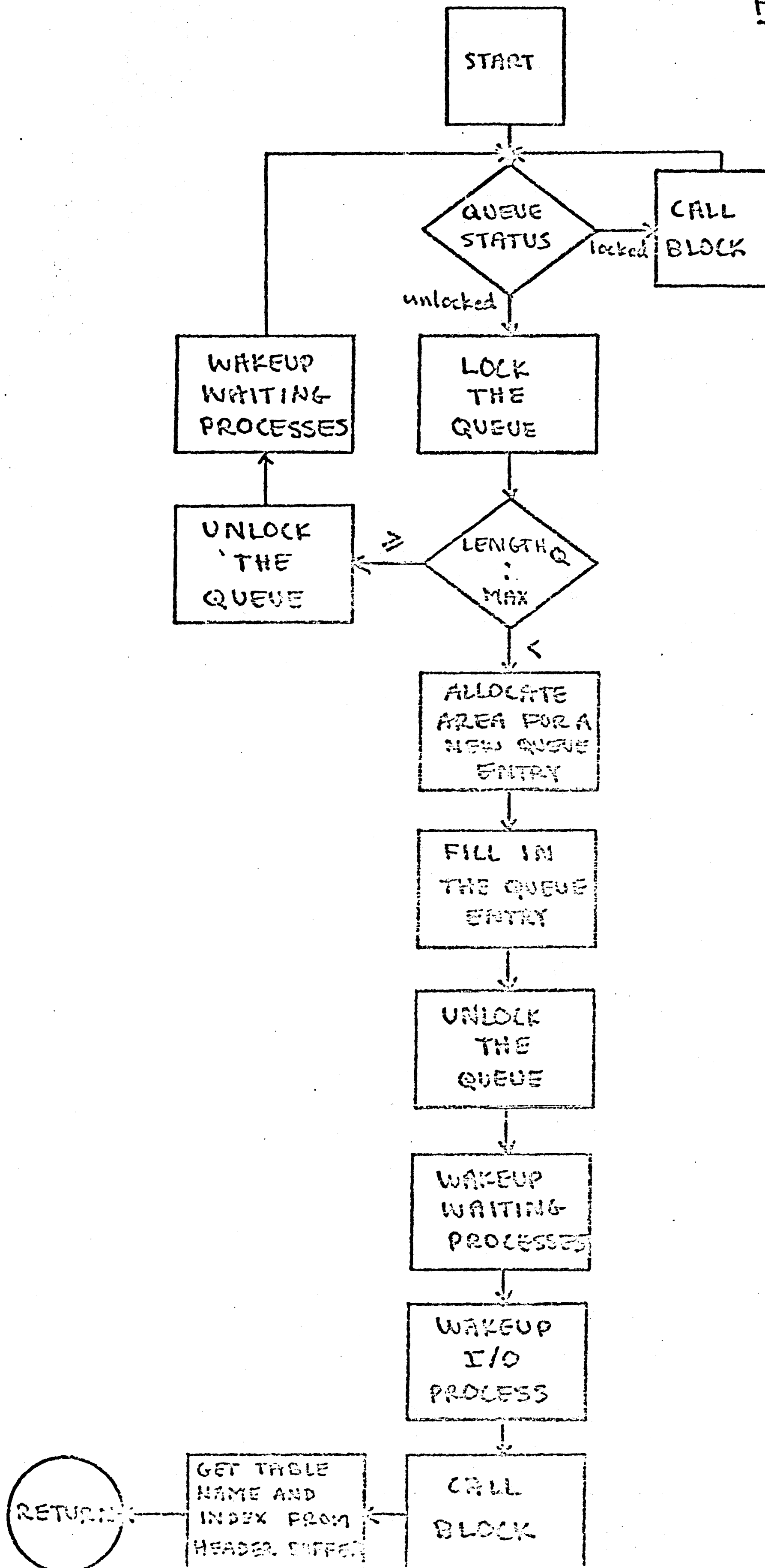
The flow of control for the I/O process is given in Figure 3.



(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.

FIGURE 2



ioroutine

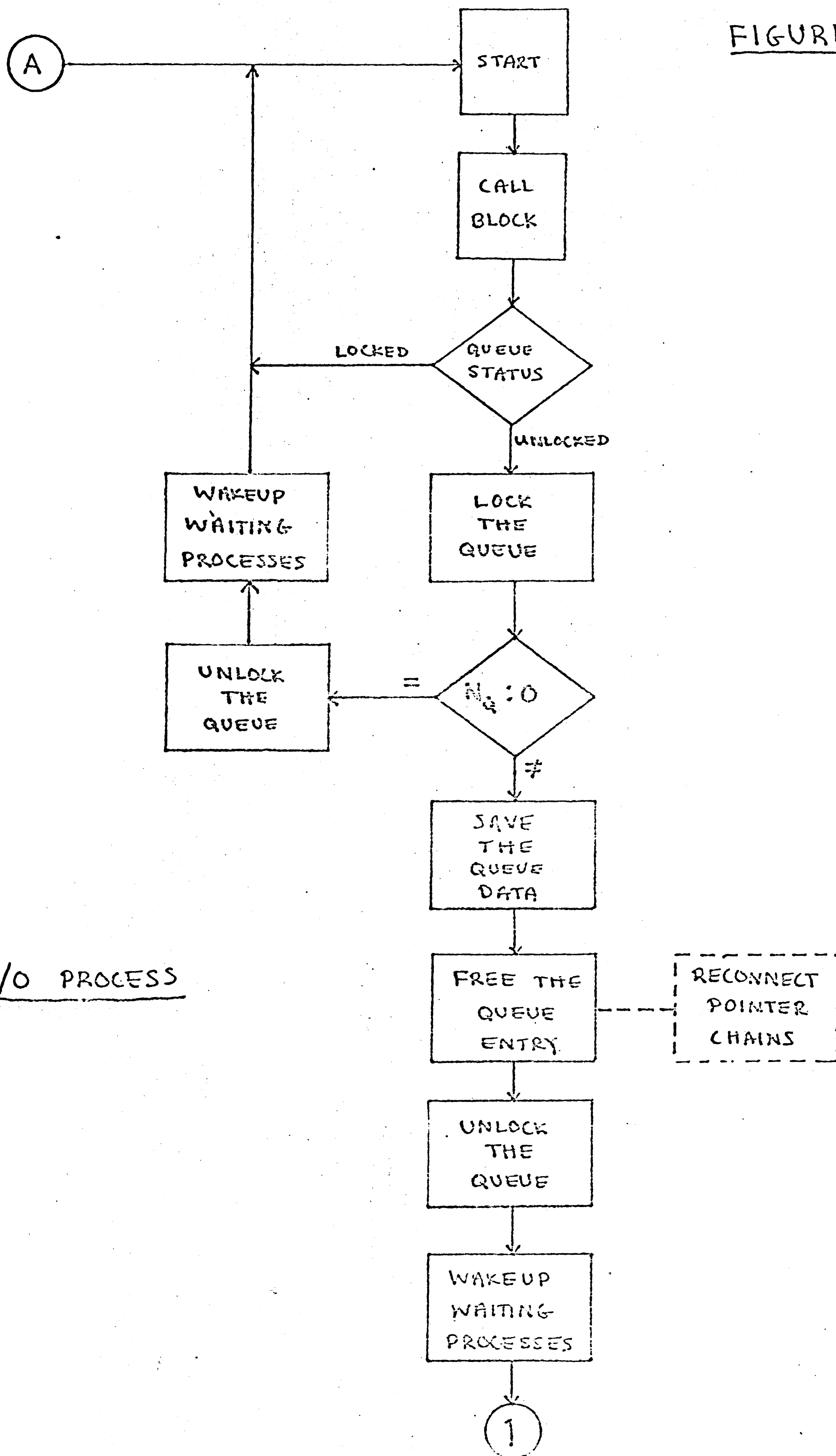


FIGURE 3
A
START
CALL BLOCK
LOCKED
QUEUE STATUS
UNLOCKED
WAKEUP WAITING PROCESSES
LOCK THE QUEUE
UNLOCK THE QUEUE
=
N_q : 0
≠
SAVE THE QUEUE DATA
I/O PROCESS
FREE THE QUEUE ENTRY
RECONNECT POINTER CHAINS
UNLOCK THE QUEUE
WAKEUP WAITING PROCESSES
1

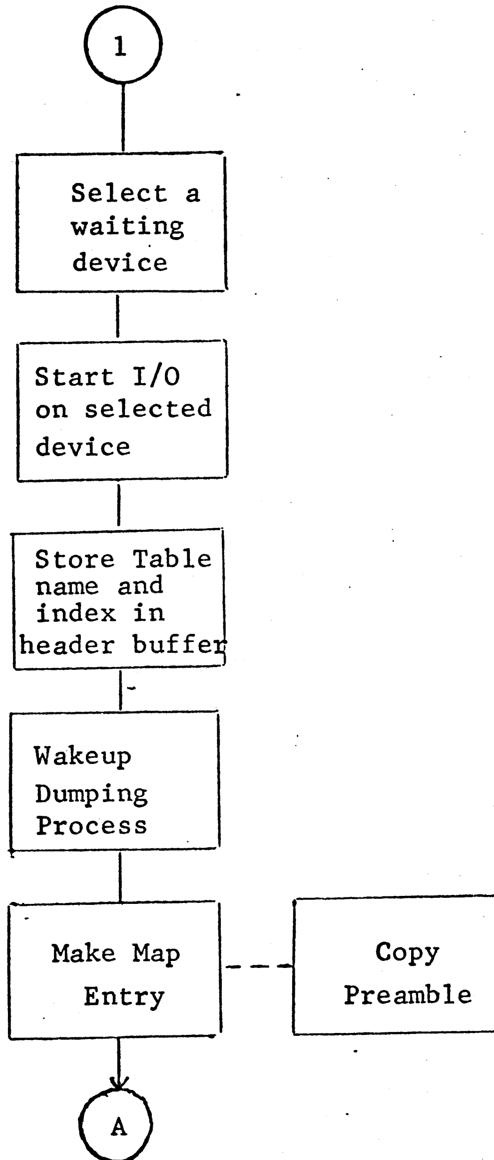


Figure 3.