To: Distribution

From: Robert S. Coren

Date: October 22, 1973

This document, which is a revision of MSB 109, describes a proposed new design for the I/O daemon. The document is divided into the following sections:

I. A brief description of the problems presented by the current implementation of the daemon

II. A summary of the ways in which the proposed design attempts to correct these problems

III. An overview of the new design

IV. A description of the Operations interface to the daemon under the new design

V. The "history" of a dprint request under the new design

VI. A more detailed discussion of many features of the new implementation, including the management of "device drivers", console input and output, and the handling of errors and abnormal conditions

VII. A description of special handling required for a remote printer/punch combination

VIII. Methods of accounting.

This document assumes some familiarity with the present implementation of the I/O daemon.

I. PROBLEMS WITH THE OLD I/O DAEMON

The present Multics I/O daemon, with one independent process per printer, has a number of basic problems. The most serious of these are:

1) All I/O daemon processes are driven by the same generalized queues. At present this results in inconvenience, since a daemon which is only running a printer can receive punch requests, and vice versa; with the advent of remote

printers, it will become an intolerable difficulty.

2) The lack of systematic communication between two or more I/O daemon processes requires removing each request from the queue *before* it is performed, to ensure that no request is performed more than once. This requires special precautiors to avoid losing requests which are in progress at the time of a system shutdown or crash.

3) As a result of problems 1) and 2) above, requests which cannot be processed when they are received (for example, punch requests received by a daemon which has not attached a punch) must be replaced in the queue by the daemon itself. This degrades priority scheduling and results in inaccurate accounting.

4) Although each daemon process can run both a printer and a punch, it cannot run them simultaneously; the printer is idle while the punch is running, and vice versa.

5) Once a request is completed -- or is believed by the daemon to be completed -- it is gone. If a printer, for example, has been having ribbon or paper problems for the past four requests, and no one (including the printer software) has noticed it, the only remedy is to resubmit the requests.

II. PROPOSED SOLUTION

The new design attempts to solve all of the abovementioned problems. The central feature of this design is the concept of *one* central I/O daemon process, called the "I/O Coordinator" (or "I/O Daemon Coordinator"), and a lot of subordinate processes called "device drivers." Each driver will run one device, and will be fed requests one at a time by the coordinator. This design meets the above problems as follows:

1) Each "device class" is fed from a separate queue (or group of priority-ordered queues). A "device class" is considered to be a pair [device type, location]; e.g., "on-site printer" would be one device class, "on-site punch" would be another, "remote printer at CISL" would be a third, etc. There can be any number of devices in a given class; each device would have its own "driver" process, but they would all be fed from the same set of queues. The dprint command will place requests in one queue or another in accordance with a new "-device_class" ("-dvc") control argument.

2) Requests are read from the queues by the coordinator only, so there is no duplication problem. A request is not removed

from the queue until after it has been completed.

3) No attempt will be made to process a request unless there is a driver (and a device) waiting for it. Besides, since requests are not destroyed until they have been performed, there is never any necessity for the daemon to replace a request in the queue.

4) In general, no two devices will be run by the same driver, so there is no reason why two essentially independent devices cannot run (effectively) simultaneously. (The special case of a remote printer-punch combination, which is not strictly speaking two independent devices, will be discussed later in this document.)

5) Each request, after completion and deletion from the queue, will be chained on to a special list for a fixed period of time (like half an hour). Such a request will be repeatable any time within the specified limit (but will not be automatically redone if the system crashes while it is in the list). Half an hour after the completion of each request, an alarm will go off, and the oldest request in the list will be deleted.

We have considered the possibility of operating the various devices from a single process, but this idea introduces several problems which are unlikely to be solved by the time that we will need the new daemon to run remote devices. In particular, such a design would entail either the development of a new asynchronous I/O interface or the introduction of subtasks within a process (or both). Implementation of both of these features is being considered for the future, and, once they are available, it will probably not be excessively difficult to adapt the design described herein to a single process.

III. OVERVIEW OF THE NEW DAEMON DESIGN

The I/O coordinator will be driven by wakeups coming from the various device drivers, indicating that they are ready for work or have just completed requests. The coordinator process will have a process-group id of IO.SysDaemon.z and will be created in the same manner as the present I/O daemon (presumably, in most cases, as part of an exec_com executed at system-start-up time). Driver processes will generally be called CLASS.SysDaemon.z, (1) where CLASS is the name of the device class the driver is running; these processes will be created by

(1) Drivers that run remote devices belonging to users may be given different project id's for accounting purposes.

operator "login" commands. Depending on the class of the driver, its process will either attach an on-site device and begin work, or wait until it receives a "dial" command drom a remote device. (When it receives the "dial" command, the driver will do further checking, such as ensuring that the device is of the correct type, and possibly requesting a password.) (1) Input to and output from these processes may be routed through the Message Coordinator.

Once a driver process exists, the running of the device will be largely automatic; operator intervention will only be required in special circumstances. Such operator commands as are required will generally be addressed to individual drivers; the most usual method for issuing such a command will be to type it on the daemon console (which might be any of several consoles) to be read by the driver when it is next ready (see below).

IV. OPERATIONS INTERFACE

(Note: This section does not include the special handling required for a remote printer/punch combination, which is described in Section VII.)

The coordinator will normally be logged in automatically in the course of system start-up, but it can also be logged in manually from any console connected to the initializer through the message coordinator, in the same manner as the present I/O daemon, i.e. by typing:

```
login IO SysDaemon io
```

but this command will be accepted only if the coordinator is <u>not</u> already logged in.

A driver process can be created by typing (again from the initializer console):

```
login CLASS SysDaemon DEV_ID
```

which will create a process to run device DEV_ID, and give it a source name of DEV_ID. (2) This process will signal the

---

(1) Presumably some of the drivers will be brought up at system-start-up time as well.

(2) These processes can also be logged in like normal users from any console; in addition, the coordinator's and drivers' process overseers will contain a "test" entry point enabling them to be tested within the regular Multics command environment.

coordinator when it is ready to run (either immediately or when it receives a valid "dial" command from an appropriate device).

The coordinator will immediately start feeding requests from the queues for that device class to the driver, which will execute them on the specified device. If the operator wishes to type further commands to the driver, he should type:

r DEV_ID COMMAND_LINE

If the driver is idle, it will receive the command immediately; if it is processing a request, it will receive the command when the current request is finished.

The command thus input can be any one of the following:

detach

this command effectively detaches the device and disables the driver. It can be used when some temporary problem arises on the device. The driver is inhibited from receiving output requests, but its process is not destroyed.

attach

This is used to undo a previous "detach" command; it restores the device to service and enables the driver to receive requests from the coordinator. it has no effect on an already attached driver.

logout

This command is used to terminate the driver process.

restart n

This command causes all requests performed by this driver which are still available, starting with the one to which it assigned the identification number n, to be redone. All devices of the same class will be eligible to perform these restarted requests, and they will take priority over regularly-queued requests.

save n

This command prevents any request performed by this driver, starting with the one it assigned the identification number n, from being deleted from the list of saved requests. (Deletion is reenabled after the request has been redone in response to a "restart".) It might be used when a "restart n" command is anticipated but cannot be issued yet (e.g., because the printer's ribbon has to be replaced first). Requests for which "restart" has been issued are automatically "save"d until after they have been redone.

If it is necessary to interrupt a driver in the middle of a request (for example, if a print request is producing reams of garbage), a QUIT should be signalled by typing:

quit DEV_ID

Printing (or punching) will be suspended, and the driver will await a further command. This command may be any of those described above, in which case the driver will behave as if the "kill" command described below had been input, and then proceed to execute the command typed. In addition, any of the following commands may be typed (again in the format "r DEV_ID COMMAND"):

kill

The current request is aborted, but is saved in the list of restartable requests. The driver proceeds to the next request (if any).

cancel

The current request is aborted, but is not saved. The driver proceeds to the next request (if any).

restart

If input without an argument, this command causes the current request to be restarted from the beginning (as if it had just been received from the coordinator).

V. THE HISTORY OF A DPRINT REQUEST

A user enters a dprint request in the same manner as before, except that an additional optional control argument ("-device_class" or "-dvc") may be included. The request will be placed in the message segment which represents the specified priority queue for the specified device class. (1) The request will remain in the queue until a driver of the right class wakes up the coordinator to say that it is ready for work.

When the coordinator receives this wakeup, it will examine the the highest-priority queue for that class. If there is a request in that queue after the one which it had remembered

---

(1) The "-device_class" control argument will, of course, have a default value, which will probably direct output to an on-site printer (or punch). The number of priority queues, instead of being fixed at three as at present, will probably be made an installation parameter.

as "last read", (2) the request is processed as described below; otherwise the coordinator repeats its check on the next lowest priority queue for the class, and so on, until either an unprocessed request is encountered or all queues for the device class have been inspected.

Once the coordinator has found a request for the ready driver, it will allocate a descriptor for the request, which includes the return arguments from the message_segment_$read call. (These arguments, in turn, include a pointer to the message itself.) Other information placed in the descriptor by the coordinator includes the device id of the driver to which it has been assigned, and the priority queue from which it was taken. The coordinator then sends the driver a wakeup over an event channel whose name was passed to the driver when it first came up.

The driver, when it receives the wakeup, will process the request in much the same way as the currently-existing I/O daemon, including access-checking and accounting. (It will not delete a file for which the "-delete" option was specified.) It will also fill in additional information in the request descriptor, including the request's sequential identification number and various status information (such as abnormal I/O status codes, indicators of whether the request was killed or cancelled, etc.). When it has finished printing (or punching) the request, it sends a wakeup to the coordinator, passing it the offset of the request descriptor in the event message.

When the coordinator receives this wakeup, it will record the clock time in the request descriptor, and thread the descriptor on to the end of a list of completed requests. (The coordinator keeps static pointers to the head and tail of this list.) It then sets an alarm to go off half an hour (or whatever other interval is chosen by the installation) of clock time later for removing the descriptor from the "completed" list. It is at this point that the request itself is deleted from the message segment. Now, if the driver has indicated that it is ready for more work, the coordinator will inspect the queues for that device class, starting with the highest-priority queue as before.

Our original request, meanwhile, is sitting in the "completed" list. If the driver that performed it receives the operator command "restart n", where n is less than or equal to the identification number of our request, the coordinator will

---

(2) The coordinator keeps a record of the unique id of the last message processed in each queue and, when called upon to inspect the queue again, simply reads the next message (if the last-read message has been deleted, the first message in the queue is read).

feed all drivers for that device class from the "completed" list rather than from the message segment queues, until all subsequent requests originally processed by that driver (including our friend) have been redone. (Descriptors marked as "restarted" are not rethreaded into the "completed" list, but are left in their original positions; nor is a new alarm set after a restarted request is finished.)

When the half-hour alarm (set at the original completion of our request) goes off, the coordinator will free the first descriptor on the "completed" list, along wiht with its associated message, unless that request is currently being redone or has been marked as "saved". (Before the descriptor and message are freed, the "-delete" option, if specified, is honored.) Since there is a one-to-one correspondence between request completions and alarms, the "completed" list will not pile up indefinitely.

If when the driver originally wakes up the coordinator on completion of a request, it indicates that the request was cancelled (by an operator "cancel" command), the request descriptor is not threaded into the "completed" list; instead, after deleting the message from the queue, the coordinator frees the request and its descriptor immediately.

VI. FURTHER IMPLEMENTATION DETAILS

1. Initialization of coordinator.

The names and characteristics of valid devices and device classes are kept in an ascii file whose format is similar to that of a bindfile. (This file must be set up prior to running the daemon.) When the coordinator is logged in, it translates the contents of this file into a set of binary tables which it and the driver processes will use to keep track of what classes and devices are available and/or running. It will also set up areas for request messages and descriptors, and make its process id available to driver processes.

2. Initialization of drivers.

When a driver process is ready to run, a special initializing procedure is invoked. This procedure will be responsible for creating an event wait channel over which the coordinator may signal the driver, as well as initiating all data segments that will be used in communicating with the coordinator. In addition, it will attach its device through a DIM whose name is in the table of device-class- and device-dependent information set up by the coordinator. Once it has done these things, it will place its device id and the name of the event channel in a reserved area (which it will lock from other drivers) and send a wakeup to the coordinator along an event call channel provided for informing the coordinator that a new driver process has been created. It then goes blocked waiting for the coordinator to finish initializing the driver description.

The coordinator, on receiving the wakeup, allocates a structure for communication with the driver, in which it places the driver's process id, the name of the channel over which it expects to be woken up between requests, and the information that was provided by the driver. (1) It sends the driver a wakeup to inform it that it has set up this structure; the driver then unlocks the area where it put its device id, etc.

3. Input to drivers

Once the driver process has been initialized, it should always be doing one of two things: 1) performing a print or punch request, or 2) waiting to be sent a request from the coordinator or a command from the operator, whichever happens first. It is not desirable for the coordinator to allocate a descriptor for a request until it knows there is a driver ready to process it, nor do we wish to send a request to a driver which has meanwhile started to process operator input. Therefore, the driver must be able to keep control of when it goes blocked for input if there is no read-ahead. It does this by utilizing the "read_status" order call to the tty_dim (see MCR #75). (2) The basic algorithm is as follows:

(1) The driver checks for input by issuing the "read_status" order call.

(2) If there is already input, the driver reads it through ios_$read and processes the command as described earlier. It then goes to step (1).

(3) If there is no input yet, the driver sends a wakeup to the coordinator indicating that it is ready for work, and goes blocked on a list of event wait channels, of which the first

---

(1) For purposes of security, each driver's structure is allocated in a separate segment, to which only the coordinator and that driver have access; this avoids access problems that might otherwise arise when non-SysDaemon driver processes are running, since the driver must have "w" access to this structure. For the same reason, when the coordinator sends the driver a request, it copies the request descriptor into the driver's communications segment.

(2) This order call will be implemented in the g115_dim as well.

is the channel returned to it by the "read_status" call, and the second is the channel over which the coordinator sends request wakeups.

(4) The coordinator will check to see if it has a request available of the device class associated with the driver. If it has one, it checks a bit in the driver's communications structure to see if the driver is still ready (since it might have received input in the meantime). If it is ready, the coordinator sets another bit indicating that it is about to send the driver a request; it then allocates a descriptor for the request and sends the driver a wakeup.

(5) If the next wakeup the driver receives is from the coordinator, it processes the request and sends the coordinator a wakeup indicating that the request is finished. It then checks to see if any input has come in; if there is any, it goes to step (2), otherwise it goes blocked again as in step (3).

(6) If the driver receives a wakeup on the channel designated for input, it proceeds to step (2) <u>unless</u> the coordinator has set the "request-pending" flag as described in step (4). In this latter case, the driver goes blocked <u>only</u> on the coordinator channel; when the wakeup arrives, it performs the request, marks itself as <u>not</u> ready, sends the coordinator a wakeup signifying the completion of the request, and proceeds to step (2).

If the driver receives a "quit" signal, it will make a normal call to ios_$read and go blocked if necessary, since after a "quit" only operator input is of immediate interest.

When the driver goes blocked as described in step (3) above, it sets a 30-second real-time timer; if this timer goes off before any other wakeup is received, the driver sends the coordinator another "ready-for-work" wakeup, and goes blocked again. This procedure has a twofold purpose: for one thing, it ensures that the driver will find out if the coordinator process has ceased to exist (otherwise, the driver might remain blocked indefinitely); for another, it causes the coordinator to check and see if any requests have been placed in the queues in the meantime. The timer is cancelled whenever the driver receives a legitimate wakeup from the coordinator or for input.

CARD READING

There is no reason, actually, that the coordinator should ever have to know about card-reading. On the rare occasions when Operations wishes to read in a card deck, an

Input.SysDaemon (or Cards.SysDaemon) process can be logged in to attach the card reader. A start_up.ec for such a process can be provided to invoke the "read_cards" procedure automatically.

The card reader attached to a remote printer/punch is a slightly special case, which is discussed in Section VII below.

MESSAGES FROM THE DAEMON

The drivers will have several different kinds of output to produce (aside from the printing and punching of user files): "normal operation" messages ("Request 75.2: Printing >doc>info>bugs.info for Girtell.Multics.a", etc.); warning or "information" messages ("Enter request:", "No punch requests in queue", etc.); and error messages (which category includes questions sent through command_query_, e.g. "Do you wish to cancel request 75?"). These three types of messages may want to go to three different places, and accordingly will be written on three different (or at least differently-named) streams. "Normal operation" messages will be written on "log_output" and will probably be directed to a file for later examination (if anyone is interested). Warning messages will be written on "user_output" and should probably appear on a terminal. Error messages will be written on "error_output" and <u>must</u> appear on a terminal. The routing of all these streams will be performed through the message coordinator; for devices in the machine room, "error_output" and possibly "user_output" for all drivers will presumably be directed to a single console, whereas for a remote printer/punch they will most likely appear on the printer. (1)

The coordinator will probably produce error messages only; these will be written on "error_output" and may be sent to the same console as the on-site drivers' "error_output". In addition, certain serious error conditions will be recorded on a "log_output" stream, which may be directed to a file for future reference.

ERRORS AND ABNORMAL CONDITIONS

The primary goal in handling errors arising during daemon operation is to ensure that the requisite information is reported while disruption of normal operations is kept to a minimum. In any case where recovery is possible, the process

---

(1) Similarly, on-site device drivers will generally read "user_input" from the same console, and remote drivers will read from the remote card reader.

which encountered the error will report it and continue its work.

1. Errors while processing a request

These will be handled very much as in the present implementation. Missing or zero-length segments, insufficient access, etc. will result in the request being skipped; I/O errors and unclaimed signals during printing or punching (e.g. out_of_bounds because a segment was truncated or deleted out from under the daemon) will cause the request to be aborted. In either case, the driver will proceed to wake up the coordinator as if it had completed the request, but it will also place a status/error code in the request descriptor. The coordinator can use this code to decide whether or not to save the request in the "completed" list, and also in certain cases (e.g. "device not attached") whether the driver is still usable.

2. Space allocation problems

If there are a large number of very busy drivers, it is remotely possible that the area used for request messages will get filled up; in this case, an error message will be written on "error_output" and in the system log. The coordinator will then free the oldest request on the "completed" list (provided it is not in the process of being redone) and try the allocation again. If this error recurs frequently, it strongly suggests some systematic problem in the coordinator's space allocation procedures. Similar considerations apply if the space for request descriptors becomes full.

3. Message segment problems

Handling of bad or inconsistent message segments will be greatly facilitated by an entry point that is being added to the message_segment_ programs which will permit a sufficiently privileged process to find out if a message segment has been "salvaged". Whenever the first driver for any given device class is created, the coordinator will examine and reset the "salvaged" bit in each of that class's queues. If in reading a request, the coordinator gets a code of error_table_$badseg (indicating that the message segment was found to be in error and salvaged during that call), it will try to read the request again. If it gets the same code, it will complain loudly (sending BEL characters to "error_output", for example) and behave as if the desired message was not in the queue. Any time that a message segment is discovered to have been salvaged, a message indicating that requests may have been lost is written on "log_output". If retries are repeatedly unsuccessful, the best bet is probably to shut the daemon down and look at the message segment to try to find out why it is unusable.

If a message that had previously been read is found to be missing when an attempt is made to delete it or read the next message after it, the coordinator will check to see if the message segment was salvaged. If it was not, it is probably safe to assume that someone removed the request (through the cancel_daemon_request command) while it was being performed.

4. Other errors in the coordinator

In the event of unexplained errors arising in the coordinator process (unclaimed signals, unrecognized event channel names, bad codes from file system primitives, etc.) a message will appear on "error_output" and the coordinator will go blocked waiting for the next event it is to service, after doing its best to ensure that its data bases are in a consistent state. Frequent occurrences of this kind probably indicate program errors in the coordinator itself.

5. Lost processes

If the coordinator, in an attempt to wake up a driver process, discovers that the driver process no longer exists, it prints an error message to this effect, and behaves as if the driver process had told it it was logging out; i.e., it frees the driver's communications structure, and removes the driver from its list of active devices.

If a driver finds that the coordinator process no longer exists, it goes to sleep for a brief period; when it wakes up it checks to see if the process id provided in the coordinator's data base has changed. If it has, the driver reinitializes itself and announces itself to the new coordinator as a new process; otherwise, it goes to sleep again, and repeats the test at regular intervals until the coordinator's process id changes.

VII. REMOTE PRINTER/PUNCH COMBINATION

This is a somewhat special case, since from the coordinator's point of view the printer and the punch are separate devices belonging to different classes, but in fact they cannot be run simultaneously. As a result, a single process will actually run both devices, but it will present itself to the coordinator as two separate drivers. The following special commands are used to control the remote device's print and punch functions:

start_print

requests that only the printer be run. If the punch was running, it will be disabled (as soon as it has finished its current request).

start_punch

requests that only the punch be run. If the printer was running, it will be disabled.

start_print_punch

requests that requests be taken from both print and punch queues, but that print requests be given priority.

start_punch_print

requests that requests from both print and punch queues, but that punch requests be given priority.

One of the four above commands must be issued to start running the device.

In addition, the following commands are provided:

stop_print *or* detach_print

stops the servicing of print requests. If the punch was active, it will continue to be so; otherwise the process will wait for further commands.

stop_punch *or* detach_punch

has precisely analogous meaning for the punch.

read_cards

indicates that the user (operator) intends to feed a deck of cards into the remote card-reader. Neither print nor punch requests will be processed while the deck is being read in; after reading, the driver process will await further commands before resuming processing.

Implementation details

After receiving and validating the "dial" command, the remote driver process awaits a command telling it what function to perform.

The first time it receives a command to start the printer, it sends the coordinator an "initializing" wakeup, as described in Section VI; it sends another such wakeup when first requested to start the punch. (1) The coordinator allocates two separate driver structures, and never notices that these two driver "processes" have the same process id.

When the driver is supposed to be processing requests for one device only, it marks the other one's driver structure as "not ready". When it has received a "start_print_punch" or "start_punch_print" command, it marks both structures "ready", and blocks on event channels for both devices. (2) As a result, wakeups for punch requests (in the case of "start_print_punch" will not be received unless there are no pending print requests. The driver responds to "stop_print" or "stop_punch" commands by taking the designated event channel out of the wait list (and promoting the other one to higher priority if necessary).

VIII. METHODS OF ACCOUNTING

For a driver process operating installation-owned equipment with a project id of "SysDaemon", accounting will be similar in effect to the present system; that is, the individual user who requested each piece of output will be charged according to the length of the file and the priority queue used. For remote devices owned or rented by customers, however, it seems more reasonable to charge the customer for CPU time, etc., used by the process running the device. If the device's driver process runs as a "user" on a project charged to the customer, this will happen automatically; the customer should nonetheless have a way of ascertaining how his I/O daemon charges are being spent.

To facilitate this, every device will have associated with it (in the I/O daemon's parameter file) an "accounting" attribute, which will be either "system" for SysDaemon-type accounting, or the pathname of a private accounting rouine. The general driver accounting routine called after completion of every output request will place in a pdt template entry all the relevant statistics for the cost of the request. If it is running a "system" device, it will pass this structure on to the system's "charge_user_" subroutine, which will increment the items in the user's actual pdt entry accordingly. Otherwise it will call the

---

(1) If the first command it receives is either "start_print_punch" or "start_punch_print", it will send these two wakeups back-to-back.

(2) In other words, the list of event channels described in Section VI, part 3, now contains three channel names: the input channel, the request channel for the device assigned higher priority, and the request channel for the other device, in that order.

private accounting routine, which can make whatever use of the statistics it sees fit.

Clearly, a driver running as a SysDaemon must use "system" accounting, since otherwise the users of the device could avoid being charged for its use altogether; therefore the driver's initialization code, which finds the specified accounting routine, will not accept a user-supplied accounting routine if ithe process's project id is "SysDaemon".