To: Distribution

From: Jerry A. Stern and Jerold C. Whitmore

Date: October 29, 1974

Subject: I/O Daemon Modifications for the Access Isolation Mechanism

Introduction

This MTB describes proposed changes to the I/O Daemon in support of the Access Isolation Mechanism. The reader is assumed to be familiar with the basic principles of the Access Isolation Mechanism, as well as the relevant terminology, defined in MTB-100.

The modifications suggested here were, for the most part, originally proposed to satisfy certain requirements of the Air Force Data Services Center. However, as with other features of the Access Isolation Mechanism, most of the new features proposed for the I/O Daemon will be of general use at many Multics sites. The following four requirements are specifically considered in this MTB:

1) It must be possible to instruct a device driver process to handle only requests of a specified range of access classes.

2) The head sheet for each printout must contain a banner identifying the access class assigned to the printout.

3) A user must be able to specify, by means of dprint command options or defaults, that header and footer labels be placed on each page of printed output.

4) Each printer driver process must be capable of preparing an "accountability form" for each piece of printed output. (In the case of AFDSC, an accountability form will be used to officially record the transmission of a classified printout to an appropriately authorized user. At other sites, forms of somewhat different format may be used for a similar purpose.)

Since the use of the above features is at the discretion of the individual site or user, no change in I/O Daemon operation will result unless desired.

In the discussion which follows, the implications of each of the above four requirements is examined and an implementation is described. Afterwards, some access control problems posed by the Access Isolation Mechanism for the I/O Daemon are investigated. Finally, a summarization of all proposed changes is presented.

Access Class Ranges for Device Drivers

It is desired that an access class range be associated with each device driver process and that only requests within this range be handled by the driver. In order to understand the meaning and implications of this idea, it is worthwhile to briefly review some features of the I/O Daemon organization and operation.

The collection of user requests into queues and the subsequent distribution of these requests to driver processes revolves around the notion of device classes. When a user submits an I/O request, he either explicitly specifies a device class or else a default device class is assumed. The device class uniquely determines a set of queues, each of which represents a different priority. Such a set of queues will be referred to hereafter as a "queue group." Each driver process is uniquely associated with a device class and hence with a queue group. Drivers of the same device class are considered to be equivalent in the sense that any one of them can handle any request from the appropriate queue group. Thus, when a driver informs the coordinator that it is ready for work, the coordinator simply selects the oldest request of highest priority from the queue group associated with the driver's device class.

With the advent of the Access Isolation Mechanism, each driver process will be assigned a specific authorization. To the greatest extent possible, driver processes will not make use of any system privileges. Therefore, if we were to allow drivers of different authorizations to belong to the same device class, these drivers could no longer be considered equivalent. A segment accessible to one of the drivers might not be accessible to another. Hence, in order to preserve the meaning of device classes, all drivers of the same device class will have the same authorization. Clearly, this authorization defines an upper access limit for the device class.

A simple way to proceed in achieving the desired access ranges for drivers is to associate the access range with the device class. Ignoring the details of this approach for the moment, only one conceptual problem is evident. Where in the current system there now exists one device class and one queue group for a category of devices, e.g., central site printers, there would be perhaps several device classes and several corresponding queue groups in the new scheme, each having a different access range.

Such an arrangement is by no means technically infeasible, but it does create inconveniences for the user and the operations staff. The user certainly does not wish to concern himself with which access range is appropriate for his request. This should and could be determined automatically by the system. However, a more serious problem arises over the fact that the access ranges associated with a device class are intended to be dynamically reconfigurable. For example, a site with three printers may ordinarily have three device classes with three different access ranges for these printers. If one printer should fail, however, it may be desirable to reconfigure the access ranges of the remaining two printers so as to process the requests formerly handled by the inoperative printer. Unfortunately, there is no easy way to accomplish this reconfiguration since the requests have already been segregated into separate queues based on the original three access ranges.

In order to solve the problem described above, it is proposed that the one-to-one mapping between device classes and queue groups be changed to a many-to-one mapping. In other words, it will be possible for one queue group to serve many device classes. Actually, it is convenient to think of the queue group as defining a "static" device class which is identical to the current notion of device class. When a user submits an I/O request, he will specify (explicitly or implicitly) the static device class. Driver processes will be associated with "dynamic" device classes, many of which can draw requests from the same static device class. Thus, whenever it is desired to reconfigure the access ranges of the dynamic device classes, no reshuffling of the queues is necessary.

Although the change described above may sound rather severe, this approach has been chosen for the very reason that it requires relatively few changes to the I/O Daemon software. As far as the relationship between the coordinator and drivers is concerned, the implementation of device classes is basically unchanged. A new parameter for the I/O Daemon parms file will be defined which permits specification of the access range of a (dynamic) device class. Also, a second new parameter will be defined which permits specification of a queue group name for a device class. When the parms file is examined during the initialization of the coordinator, all device classes sharing the same queue group will be threaded together. Furthermore, a new data base, called the queue group table, will be constructed which contains one entry for each queue group. Each entry will have a pointer to the head of the threaded list of associated device classes as well as pointers to (or indexes of) the message segments in the queue group. Each device class entry will contain a pointer to its associated queue group entry.

Aside from the extra initialization described above, only one other section of the I/O Coordinator will require significant

modification. (Note that no changes to the drivers are necessary to implement access class ranges.) The subroutine responsible for reading requests from the queues, called find_next_request_, must understand the device class to queue group mapping. When given a device class, find_next_request_ will ascertain the appropriate queue group and read the oldest request from highest priority non-empty queue (as it does now). It must then determine if the access class of the request message is within the access range of the specified device class. If so, the request is returned as usual. If not, find_next_request_ will scan the threaded list of device classes for the queue group until finding a device class with the proper access range. The message ID of the request will then be added to a "waiting list" for that device class. The reading of messages, and the adding of these messages to waiting lists, will continue until a message is found within the access range of the specified device class or until the queue group is exhausted. Thus, it can now be seen, that the algorithm followed by find_next_request_ is to first check the waiting list for a device class and, if this is empty, to then begin reading messages from the associated queues.

The effect of the above scheme is to delay the binding between a request and a dynamic device class until the moment the request is read from the queues. Furthermore, this binding can always be reconfigured, even for requests in the waiting lists. This is accomplished by simply changing the parms file and then reinitializing the coordinator. The old waiting lists are discarded and new ones are created for the new dynamic device classes. No juggling of the queues is ever necessary. Note also that at installations which continue to maintain a one-to-one correspondence between queue groups and dynamic device classes, no requests will ever be added to a waiting list.

Access Class Banners

Just as the access class stored in a branch is used internally to protect segments, so too will the access class banner on a head sheet be used externally to protect printouts. The access class banner provides an administrative control over the distribution of printouts which supersedes the existing discretionary controls (i.e. person and project name banners).

A general rule of the Access Isolation Mechanism dictates that an object is assigned an access class equal to the authorization of the process that created it. A strict interpretation of this rule would suggest that the access class assigned to a printout, i.e., the access class banner, should equal the authorization of the driver process that created it. Unfortunately, this scheme would result in widespread over-classification of printouts since the driver process authorization is always at the top of the access range of requests handled. Although some sites might be

willing to accept this drawback in the interest of maximum security, it seems likely that most sites would find it extremely objectionable. Since the driver process is really just a trusted intermediary which creates a printout on behalf of a user process, it seems logical, and a great deal more practical, to choose the authorization of the requesting user process as the access class for a printout. In order to satisfy those sites which may prefer the more conservative choice, a new parameter will be defined for the I/O Daemon parms file which allows an installation to specify a minimum access class banner for each device class. If this parameter is not specified, the default minimum will be the bottom of the device class access range.

The new format for a head sheet will include a third line of "big letters" containing the printout access class. Actually, a single big-letter line cannot be expected to hold an arbitrarily long access class string. Therefore, only the first component of the access class string will be printed in big letters. Beneath this, the full access class will be printed in regular type. This implies that at sites using sensitivity levels, the access banner will be a level name. At sites using categories but not levels, the access banner will be the first category name. However, if an access class string is null, as might be desired for the system low access class, then no access class banner will be printed. This implies, of course, that at sites using neither levels nor categories, the access banner will always be omitted.

Page Labels

The requirement for page header and footer labels to be added to printed output by the I/O Daemon stems from the need to place access class labels on each page of certain printouts. However, it is intended that this feature be generalized to allow a user to supply any arbitrary character string for the labels. This kind of feature has actually been considered before outside the context of the Access Isolation Mechanism. The dprint message format already provides space for a page header string, although the mechanism itself has not yet been implemented.

Several options will be added to the dprint command to support the page label feature. If the user simply wishes to use the segment access class for the page label, he will specify the "-access_label" option. If the user wishes to supply his own label he will specify the "-label" option followed immediately by the label string. If neither of these options is specified, then no labels will be added unless the site has chosen to add labels by default. This will be indicated by a new parameter in the I/O Daemon parms file. The effect of this default labeling will be an implicit "-access_label" option for all dprint commands

issued. However, a user can override the default label with the "-label" option or can request no labels by specifying the "-no_label" option.

Implementation of the labeling feature would best be accomplished by providing a new order call to the printer DIM for specifying labels. This, in turn, would require modifications to the printer DCM which does essentially all of the work for the printer DIM. It is intended that the labels be placed in the top and bottom margins of each page so as not to disturb the format of the output. Because a number of printer DIM enhancements are already in progress, it will most likely not be practical to begin work on the label feature in the very near future. Therefore, in order to meet the deadline for delivery of this feature to AFDSC, an interim solution may be adopted. A new IOSIM can be provided for the printer driver process which, when spliced in before the printer DIM, will insert labels. By use of the "noskip" mode in the printer DIM, labels can still be placed in the top and bottom page margins as desired. Obviously, this second approach is less efficient than the first and therefore will only be used temporarily if at all.

Accountability Forms and Driver Control Terminal

The requirement for accountability forms is primarily to provide a means of recording and controlling the distribution of classified output. It also serves a direct security function in the separation of output. The distribution staff can check to be sure that there is one piece of output (e.g., listing, card deck) for each accountability form. This check will prevent a malicious user from imbedding headers and trailers within his data which would fool the distribution staff into believing a phoney access class banner. A separate terminal from the current daemon console must be used to prepare the accountability forms and it should be located near the associated device.

A byproduct of the accountability form terminal is its ability to also function as a driver control terminal. The usefulness of a driver control terminal stems from physical hardware arrangement. Some sites locate one or more line printers (or other I/O devices) in physically separated areas from the central computer. However, the daemon driver console must remain in the central computer room to prevent privileged access from falling in the hands of untrusted personnel. On the other hand, the local device operator is in the best position to determine which requests should be restarted, etc. Another terminal physically located beside the device could allow the device operator to enter benign operational requests without compromising security and without requiring assistance from central operations. The use of this control/accountability form terminal would, of course, be at the option of the site.

To implement this new feature we will add a new per device class parameter to the IO daemon parms file which indicates whether a control terminal is required for the driver. The default for an unspecified parameter will be "not required." When the terminal is not required, the driver process will operate exactly as it does today.

When a control terminal is required, the driver will wait for a terminal to be dialed to the process before telling the I/O coordinator that it is ready to process requests. However, the current implementation of the dial command is too restrictive to be useful in this context. It only allows one instance of a process_group_id to request dialed devices. Under the current implementation, drivers and the IO coordinator are logged in as IO.SysDaemon. Hence, we must implement the changes to the dial command suggested by T.H.VanVleck in MTB 013.

During normal operation of the driver, the control terminal will print one accountability form for each copy of requested output from the driver process. The form may contain information which describes: the requestor, header and destination options, sequence number, banner access class, date-time, installation, pathname and access class of segment. (Note: The module which formats the output to the control terminal will be site replaceable. The normal module will print the same information provided by the I/O Daemon today which does not require a form.)

A "start" command must be issued from the control terminal before processing will begin to allow the device operator to align the accountability forms being used. A command to print a sample form will be provided for this purpose. Since the output to the control terminal may be formatted to preprinted forms, commands may not be entered without destroying the alignment. Therefore, commands will be honored only after the device operator presses "quit" on the control terminal. This allows for realignment before resuming operation (we will reset the write buffers).

The control terminal will never be allowed to enter arbitrary commands for security reasons. Also, we must restrict the set of commands, normally acceptable to the driver, which may be entered from this terminal. Specifically, the commands return, debug, detach, attach, and reattach will not be honored from the control terminal. The other commands will not create security problems (i.e., start, cancel, kill, restart, save, reinit, logout, sample (new)).

We don't want to remove the site operator's ability to control the driver. Therefore, when the driver expects input, it will first look for commands from the master driver console and then from the control terminal. (Control terminal quits will be

disabled while the master terminal has control of the process). The master console will also be able to indicate that further input and quits from the control terminal be accepted or rejected.

If the control terminal gets disconnected, the master console will be notified and the driver will wait for instructions. The operator may request that the driver continue without the control terminal or that the driver wait for another dialed terminal (reinit).

A remote driver which communicates to a device over high speed phone lines will also be able to utilize a control terminal. This, of course, would require a second phone line. Driver commands may be input from the control terminal as described above. Commands which may be entered from the remote device itself (e.g., from card reader) must be subject to the same restrictions as commands from the control terminal for security reasons.

Access Control Considerations

The preceding sections described changes to the I/O Daemon to support certain new features. This section, however, primarily describes changes necessary to cope with the impact of the Access Isolation Mechanism on the I/O Daemon environment. Also, an existing security problem is discussed.

The I/O Coordinator, by its very nature, cannot operate strictly within the rules of the Access Isolation Mechanism. Since it handles information of all access classes, it will run with a system-high access authorization. In order to send wakeups to driver processes, it will have the ipc privilege flag enabled. In order to create and modify segments of varying access classes, it will make use of privileged access to segments and directories. In order to read and delete messages of all access classes, the coordinator will have privileged access to message segments.

Several segments exist in io_daemon_dir which hold messages and message descriptors read by the coordinator from the message segment queues. Since these messages will range in access class up to system high, they must be protected in a system high segment after extraction from the message segments. Therefore, a subdirectory of io_daemon_dir will be created having a system-high access class. In this directory the coordinator will create the request_seg (used to hold messages), the req_desc_seg (used to hold message descriptors), and the new waiting list segment.

Unlike the coordinator, driver processes are, for the most part, well-suited to abiding by the restrictions of the Access Isolation Mechanism. Therefore, a number of minor changes will be made to the I/O Daemon to avoid the unnecessary use of special access privileges.

The current scheme for initializing driver processes will require slight modifications. Each driver process attempts to verify that a coordinator process does, in fact, exist by locking a coordinator lock kept in a special segment. If the lock is found to be validly locked, then a coordinator exists. However, if a driver succeeds in locking the lock, then no coordinator exists. Unfortunately, locking the lock means writing in the segment. Since drivers will have differing authorizations, they cannot all write in the same segment. Therefore, the drivers will instead copy the lock to a private data area and then attempt to lock the copy. This works even better than the present scheme since it eliminates the need for a secondary lock now used to prevent interference among drivers.

The initialization of driver-coordinator communication will also require some small changes. All drivers create a temporary "communication" segment containing information for the coordinator in io_daemon_dir. Due to differing driver authorizations, this will no longer be possible. Therefore, these temporary segments will instead be created in each driver's process directory. Upon receiving a "new driver" wakeup from a driver, the coordinator examines the communication segment, validates the driver, and then creates a "driver status" segment used for future communication. The driver status segment, now created in io_daemon_dir, must be writable by the driver process and therefore must have an access class equal to the driver's authorization. Since driver status segments of differing access classes cannot coexist in a single directory, the coordinator will create a separate upgraded subdirectory in io_daemon_dir to hold each driver status segment.

As mentioned above, messages and message descriptors will be stored in segments of system high access class and hence will not be accessible to all drivers. Message descriptors are already copied to the driver status segment by the coordinator each time a driver is given a request. Currently, the driver reads the message itself directly from the request_seg. Since this will no longer be possible, the message will also be copied to the driver status segment by the coordinator at the same time as the message descriptor.

To this point, every effort has been made to ensure that driver processes would not require the use of any special access privileges. Unfortunately, there are two cases in which the use of such privileges seems unavoidable. Following each dprint, a driver process executes a program called "charge_user_" which

updates accounting information in the pdt entry of the requesting user. Since pdt segments have system-low access classes, yet driver authorizations may range up to system high, it will be necessary for the drivers to obtain privileged access to pdt's. The other circumstance in which special access is required is within the message routing DIM. All daemons attached via the message routing DIM must write in a common segment. Therefore, mrdim_ will be modified to detect the need for special access and to attempt to obtain special access.

A security problem exists due to the fact that the coordinator process ID and event channel ID are stored in a segment accessible to all processes. This makes it possible for any process to impersonate a driver, i.e., to drain requests from the queues and to issue various commands to the coordinator such as "restart." This problem is easily corrected simply by setting the ACL of the segment containing the coordinator event channel ID to deny access to all but IO.*.* .

Detailed List of Changes

A. For Access Ranges

1. Change iodc_$init to create the queue group table/waiting list segment and to store a pointer to this segment in iodc_static.

2. Change iodc_parse_parms_ to recognize the new "access_range" and "queue_group" keywords. Initialize the queue group table and thread together device class table entries of the same queue group. Place in each device class table entry the offset of the associated queue group table entry.

3. Change iodc_$new_driver to check if a new driver is the first of its device class and if this device class is in turn the first of its queue group. If so, open the message segments in the queue group.

4. Change find_next_request_ to use the queue group table and to manage the waiting lists as described.

5. Change save_request_ to use the queue group table to determine from which message segment a given message should be deleted.

B. For Banners:

1. Change head_sheet_ to print the access class banner.

C. For Page Labels:

1. Change the dprint command to recognize the new -access_label, -label, and -no_label options.

2. Change iodc_parse_parms_ to recognize the new "label" parameter which causes labels to be added to printouts by default.

3. Change output_request_ to check for the label option and to make the appropriate order call if it is requested.

4. Change printer_dim_ to recognize a new "label" order call and to pass this on to the printer DCM.

5. Change printer_dcm_ to recognize the label order call and to insert labels in the top and bottom page margins.

6. If changes 4 and 5 cannot be made soon enough to meet the delivery deadline, then implement a new IOSIM to add labels as described.

D. For Accountability Form/Device Control Terminal:

1. Change iodc_parse_parms_ to recognize the "control_terminal" keyword.

2. Change iodd_static to hold control terminal attachment data.

3. Change remote_$init and io_daemon_driver_ to attach control terminal if required.

4. Change iodd_quit_handler to conditionally recognize input from control terminal and implement sample command.

5. Change input_cmd_ and remote_ to separate commands from master and control terminal.

6. Change output_request to call accountability form printing module if a control terminal is attached.

7. Change the answering service dial facility per MTB 013.

E. For Access Control Considerations:

1. Change iodc_init to enable the necessary special access privileges for the coordinator. Create a system high directory in which to place request_seg, req_desc_seg, and the queue group table segment.

2. Change iod_overseer_ to copy the coordinator lock before testing it for a driver process.

3. Change driver_init_$signal to create the driver_comm segment in the process directory and to store the process authorization in the driver_comm structure.

4. Change iodc_$new_driver to create an upgraded directory and hold each driver status segment.

5. Change iodc_$driver_signal to copy each dprint message to the driver status segment.

6. Change charge_user_ and mrdim_ to use privileged segment access as described.