To: Distribution

From: Steve Webber and Melanie Weaver

Subject: Run Units

Date: March 2, 1977

Many languages (PL/I, COBOL, and FORTRAN, for example) are defined in terms of a <program>, which in Multics terminology is a set of external procedures with any associated internal and external subroutines. An environment for the <program> must be created and communication among procedures within the <program> must be well-defined before the <program> is executed or "run." In Multics today, the environment used is the Multics <process>. This MTB describes a more restricted, controlled environment called a <run unit> that is better equipped to run <program>s more as the given languages specify they should be run. There are still some unresolved issues; the authors would appreciate feedback.

PURPOSES OF A <run unit>

The prime reasons for a <run unit> are listed below:

1. To isolate the name scope of external variables,

2. To cleanup (storage, file openings, etc.) after the <program run> whether it was <terminated> normally or abnormally,

3. To allow language-defined semantics related to <main program>s to be honored,

4. To protect the <program run> from earlier actions in the <process> that should have no bearing on the <program>, and

5. To guarantee newly initialized static storage (including common storage of FORTRAN).

MAIN <program>s

The COBOL and FORTRAN languages incorporate the concept of a <main program>. The semantics of certain statements in these languages changes if the program is the <main program>. Some mechanism must be provided that allows a user to declare which procedure is corsidered to be the <main program>. A <run unit> easily provides this capability.

THE <run unit>

A <run unit> is analogous to a subset of a full Multics <process> (on a per-ring basis) that can be set up, executed, and cleaned up with little lasting effect on the rest of the <process> other than changes to permanent <file>s. Items of particular interest are:

1. The use and management of reference names,
2. The allocation of <variable>s in <free storage>,
3. The allocation of segment numbers,
4. The opening and closing of <file>s,
5. The handling of external and internal static storage,
6. The handling of temporary segments,
7. The allocation of linkage sections and the snapping of links,
8. The use of search rules, and
9. The effect on debugging.

Each of these is discussed in detail after an overview of the <run unit> structure and its initialization.

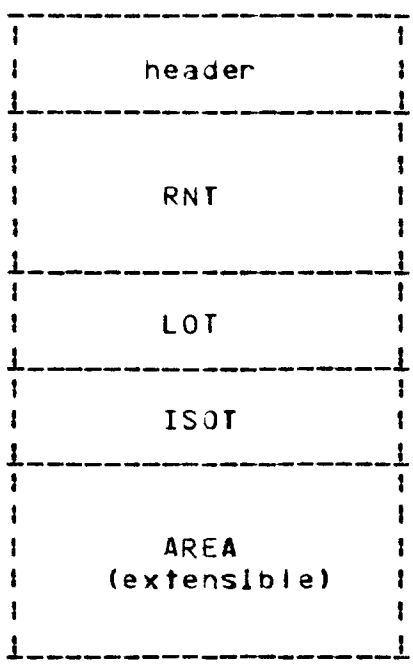
THE <run unit> ORGANIZATION

In creating a run unit several data structures must be created. These include:

1. A new LOT (linkage offset table),
2. A new ISOT (internal static offset table),
3. A new combined linkage area, and

4. A new RNT (reference name table) including search rules.

The <run unit> manager must create the above data structures. The proposal is to place all of them in a (temporary) segment organized as follows:



The initialization of each of these regions is described below:

1. The <run unit> header contains information needed by the <run unit> manager. It includes items such as a version number, pointers to the various regions, and times.

2. The RNT region is a standard RNT area as managed by the supervisor. The region is pointed to by a stack header variable (its previous value is saved either in the header or in the stack frame for the <run unit> manager). The RNT includes search rules that are, by default, copied from the RNT in effect just before the <run unit> is established.

3. The new LOT may have to be carefully set up because of the coding practices used by many system routines. In particular, system routines that use internal static storage to manage per<process> data structures, such as iox_, must be the same in the <run unit> as in the

<process>. However, most procedures will need to use the <run unit>'s name space and search rules and so will need to have their links set to a virgin state at the beginning of the <run unit>.

The proposed action taken when generating the <run unit>'s LOT and ISOT is as follows:

A. If a LOT slot is used by a "special" system routine, e.g. iox_, copy the LOT value and the ISOT value directly.

B. Otherwise, if the LOT slot is nonzero and not faulted, fill in the LOT and ISOT slots with a faulting packed pointer which yields the virgin linkage or static when needed.

C. If the LOT slot is nonzero and faulted, indicating the segment number has been used but no linkage (or static) allocated, copy the LOT and ISOT slots directly.

D. If the LOT slot is zero, set the LOT and ISOT slots to zero.

These schemes assume a lot_fault and isot_fault handler exists, although in most cases the linker will check to avoid faulting. After the LOT and ISOT have been so initialized, the only slots with valid packed pointers are those for the "special" system routines.

A more detailed discussion of the problem of "special" system routines is given below. Briefly, they will probably be changed to use some more global programming standard rather than just be named on a list in the <run unit> manager.

4. The (extensible) area at the end of the <run unit> segment is merely initialized as such. It includes linkage sections, static sections, controlled storage, external variables (common blocks and PL/I externals--*system link data), and hcs_$assign_linkage storage.

THE MANAGEMENT OF REFERENCE NAMES

When a <run unit> is created an RNT is established. By default, the only reference names initiated are those for the "special" system routines but two mechanisms exist for overriding this. The first is to copy all reference names initiated before the <run unit> is initialized into the new RNT. The second is to

specify a select subset of reference names to copy. The method used to affect this is given under the description of the use of the run command.

During the lifetime of the <run unit> all reference name activity is in relation to the new RNT. When the <run unit> is terminated the stack header is changed so that the old RNT once again takes effect. This means that all RNT activity during the <run> is forgotten when the <run unit> is terminated.

THE USE OF <free storage>

A normal Multics process has two pointers in the stack header used by the PL/I allocate statement. The first pointer points to an area used by system routines. This pointer is the one returned by get_system_free_area_ and is not replaced for the <run unit>. The second pointer is a pointer to an area into which controlled storage will be allocated and into which allocations with no in clause will be performed. When the <run unit> is initialized, this pointer is saved and then replaced with a pointer to the extensible area in the initial <run unit> segment. (Usually these pointers point to the same area.)

External variables (*system link data) currently make use of both of these (logically separate) areas but will be changed to use only the user area. When the <run unit> is initialized, the stack header variable pointing to the control information is saved and replaced by a null pointer. Hence, all external variables referenced and allocated during the <run> are isolated in the <run unit> data bases. When the <run unit> is terminated, the stack header variables are reset to their original value thereby reverting all external variables to their previous state.

THE ALLOCATION OF SEGMENT NUMBERS

One of the actions taken during a <run> is the implicit or explicit initiation of reference names and the making known of segments. In most cases, any segments made known during a <run> are not needed after termination of the <run unit> and hence it would be desirable that such segments be made unknown when the <run unit> is cleaned up. However, today there is no deterministic method of finding out why and on whose behalf a segment is made known. In light of this, the following is proposed:

> Use the LOT as a user-ring visible record of which segment numbers have been used by a process. Use any changes in the LOT during a <run> as an indication that

one or more segments were made known during the <run>.

The proposed method of cleaning up segment numbers at the end of a <run> is then to make unknown any segments recorded in the per<run unit> LOT that are not recorded in the LOT being used before the <run unit> was initialized. This requires the following proposed change to the system:

> When a segment is first made known, fill in the LOT entry in the appropriate ring with a coded, faulting packed pointer value. (This is overridden with a LOT pointer if the linker subsequently allocates storage for a linkage section.)

When a <run unit> is cleaned up, all segments with a LOT entry in the new LOT, not in the old LOT, are terminated. This is done via a call to term_$seg_ptr which causes all links to be unsnapped--even in the inherited linkage sections of "special" programs. Note that segments known in other rings are not made unknown. Segments known in the <run unit>'s ring are made unknown only if the old LOT has a zero value corresponding to the given segment.

The actual changes to the system for this would be as follows:

1. Change makeknown_ to return a status bit indicating that the usage count for the given ring changed from 0 to 1.

2. Change makeunknown_ to return a status bit indicating that the usage count for the given ring changed to 0.

3. Change initiate (ring 0 version) to set the LOT slot in the calling ring to a nonzero (faulting packed pointer) value when the usage count goes from 0 to 1.

4. Change terminate (ring 0 version) to zero the LOT slot for the given ring when the usage count goes to 0.

Note that we plan to eventually move the initiate and terminate routines from ring 0 to the user ring and this is why the particular choice of implementation was made. When these routines are in the user ring, then only user ring code manipulates the LOT for that ring. Note further that this corrects a problem currently in fs_search which touches each ring's LOT (usually 1 and 4) whenever the linker makes a segment known for the first time.

THE OPENING AND CLOSING OF <file>s

One obvious purpose for a <run unit> is to reset <file>s to their state prior to a <run>. This means leaving a <file> attached if it was attached, leaving it open if it was open, etc., but closing and detaching when appropriate. In order to do this the <run unit> manager must call any appropriate <file> management routines when the <run> is terminated. Further, the <file> management routines must be written so that they can remember the state of <file>s when first referenced during a <run>. In fact, this is just how the language I/O routines are currently designed.

Any programs that want to gain control at the end of a <run> (such as <file> management routines) must call a special entry in the <run unit> manager program indicating this desire and specifying which entry to call at <run unit> termination time. This is completely analogous to the method used today for the "finish" condition. In fact, the "finish" condition will also be signalled at the end of a <run unit>. (An info structure will be passed with it so that a run unit can be distinguished from <process> termination.)

The <run unit> manager must also be prepared to handle calls to set up such call-back requests even when a <run> is not in progress--i.e. in a <process> before a <run unit> is initialized, or after a <run unit> is terminated. (This is, of course, the only way to run things today.) In this case, for compatibility, the standard system-supplied "finish" handler calls the <run unit> manager to perform the job of calling all programs that indicated they were to be called. This preserves the concept of a Multics <process> being a <run unit> itself.

Note that any <file> activity by programs not notified at <run unit> termination is <u>not</u> cleaned up by the <run unit> manager. This means that if a <file> attachment is changed by such a program during a <run unit>, it will remain changed after the <run unit> terminates.

THE HANDLING OF EXTERNAL STATIC STORAGE

With each <run unit> is associated an RNT used in resolving external references. This RNT can be initialized in several ways ranging from containing reference names for only the "special" routines, to initially a copy of the RNT just prior to the <run>. The RNT is used by the linker to resolve external references other than *system links. *system links are resolved to a generation of storage within the <run unit>. Hence, all *system variables (common blocks, PL/I external variables with no $ in

the name, etc.) are local to the <run unit> and therefore reinitialized each time a new <run unit> is established. This is what is required for PL/I <program>s and FORTRAN <program run>s. (All COBOL working storage is similar to PL/I "internal static" and therefore does not fall into the category of external.)

THE HANDLING OF INTERNAL STATIC STORAGE

Internal static storage is reset to its initial state within the <run unit> for programs in the <run unit>. There are several system routines currently using static that would not work correctly if their static sections were reset; they must be either special-cased or recoded. They can be divided into two categories: one group is necessary for a smoothly running process and the other group maintains the command environment.

The first group includes:

```
iox_
ipc_
timer_manager_
get_temp_segments_
get_system_free_area_
```

The other list will probably include:

```
listen_
cu_
rest_of_cu_
ios_
print_ready_message_
debug
probe
progress
abbrev
abs_io_
general_ready
```

This list is subject to change, but unless most of these are converted, absentee will not work properly with <run unit>s, desired ready messages and abbreviations will disappear should the user get to command level within a <run unit>, it will not be possible to release past a <run unit> manager frame, and it will not be possible to debug across <run unit> boundaries.

There are at least three possible alternatives, each with its advantages and disadvantages. The first alternative is for the <run unit> manager to have a list of procedures whose static

is not to be reset, i.e. whose LOT and ISOT entries are to be copied. The advantages are that it is fast and does not involve coding changes or updating the linker. The difficulty with it lies in how to specify the list. In order for users to be able to modify it, it either must be an external segment or there must be an entry in the <run unit> manager for adding items. If the list contains pathnames, there is little user flexibility. If the list contains reference names, and <run unit>s can be recursive, different name spaces are used, which defeats the per<process> purpose. Also the <run unit> manager might have to check several of the possible reference names for some routines.

The second alternative is to have another storage class, internal perprocess static, which would really be only a way to tell the <run unit> manager what static to reset. This might be implemented by a new procedure option, perprocess, which would cause a new object map flag to be set (and/or the flag could be set by a command). The linker, when setting up an active linkage section, would turn on a new flag in the active linkage header. When the <run unit> manager initializes the new LOT, it would look at all the active linkage headers to determine which LOT/ISOT entries should be copied and which reset. This would involve only very minor changes to the compiler, assembler, binder and linker and would be relatively efficient. The problems with this scheme are that there would be extra paging while touching all the linkage headers and that some programs may want both per<process> and per<program> internal static.

The third alternative is to have per<process> variables. In PL/I, external variables whose names begin with "$" would have a new type of link which would be implemented like *system links (variables allocated and found by a control structure pointed to by a new pointer in the stack header). This pointer would not be reset by the <run unit> manager and a pointer to the area used would be stored either in the stack header or in the control structure. The <run unit> manager would then have to special-case only lox_ (which has definitions to static). The compiler, assembler, linker and binder would have to be updated, although not extensively. This method would be the most flexible but would be more costly when the links are first snapped and there would have to be strict naming conventions to prevent conflicts, e.g. ipc's per<process> variables would begin with "ipc_". Both of these problems would be minimized if each procedure concerned had just one per<process> variable, a pointer to a structure containing the rest of the information. However, this would involve more recoding. This alternative has the additional overhead of resetting more static and linkage sections.

THE HANDLING OF TEMPORARY SEGMENTS

The temporary segment manager, get_temp_segments_, is treated as a per<process> program and hence has per<process> internal static. This means that, as with all standard programs, appropriate clean up strategies must be followed by programs that are part of a <run unit> in order to insure that temporary segments are released at the appropriate time.

THE ALLOCATION OF LINKAGE SECTIONS

All linkage sections allocated by the linker during a <run> are allocated in the per <run unit> segment(s) via the pointer in the stack header. This pointer is set to the (extensible) <run unit> area at the initiation of the <run unit> and reset to its previous value when the <run unit> is terminated. The analogous pointer to the internal static allocation area is treated similarly.

The linker itself is driven off of these stack header pointers as well as the LOT pointer, the ISOT pointer, the RNT pointer, and the *system link pointer, all of which are changed when a <run unit> is initiated.

THE USE OF SEARCH RULES

As stated earlier, a new RNT is created with a <run unit>. The default contents of this RNT are the search rules in effect just prior to the <run>. However, it is also possible to specify alternate search rules to take effect during the <run> when the <run unit> is initiated. (See the description of the run command below.)

THE EFFECT ON DEBUGGING

The debuggers' static sections will be special-cased or recoded so that they can operate across <run unit> boundaries, and there should be no difficulty in tracing the entire stack. However, it will be more difficult to debug programs with stack frames before the <run unit> because the environment pointers in the stack header will not be appropriate. This affects such things as the debug &l request, finding values of external variables, and reprint_error. To enable debuggers to handle this situation, it is proposed that there be a new stack frame flag indicating a <run unit> manager frame and an entry in the <run unit> manager which will return the stack header environment information for a given stack frame. Not all debugging tools

will be changed to take advantage of these features initially. One or two subroutines, such as get_link_ptr_, will have additional entry points that take a stack frame pointer for the benefit of programs such as trace stack.

Name: run

The run command initiates a run unit in which to execute a program. The effect of executing a program within the constraint of a run unit rather than not so constrained is that the run unit isolates the local effects of the program run, such as linking to other programs, initiating reference names, opening files, etc., to the environment of the run unit and, hence, upon run completion, the user's process appears as it did before the run. The resources managed by the run unit manager include reference names, linkage sections, files, segment numbers, static storage and external variables (including FORTRAN common blocks). All of these are reset to their prior state when the run unit is terminated.

Usage

run {control_args} main_program {command_args}

where:

1. control_args
   are used by the run unit manager to control and initialize the environment for the run. These must all precede the command_name, and may be chosen from the following:

   -limit n
   sets a bound of n seconds of virtual CPU time for the run. The default value for n is infinite.

   -use path
   directs the run unit manager to initiate all names on the given segments as reference names in the (new) RNT to be used during the program run. If any of these names already exist in the new RNT, the given names replace the older names. Path is either the name of an object segment or the pathname of a file containing names of segments.

   -search_rules path
   directs the run unit manager to use the search rules whose ASCII representation is in the segment named by path. The search rules are set up exactly in the order given, and the keywords accepted by the set_search_rules command are honored.

   -copy_rnt
   indicates that all reference names in the (old) RNT in effect just prior to the run are to be copied into the (new) RNT to be used during the run. Any reference names initiated via the -use control argument override other reference names. The

default is for the (new) RNT to contain only the names initiated via the -use control argument.

-common path
directs the run unit manager to assume path is the pathname of a block data subprogram that includes the initial values for variables in FORTRAN common storage. If no -common control argument is specified, common blocks are initialized when first referenced. This means that a block data subprogram must be compiled in (or bound in) with the first program to reference the common in the run.

2. main_program
is the name of the main program of the run. The main program name is the first non-control argument on the command line.

3. command_args
are arguments and control arguments to be passed to the main program of the run. These must all follow the main program name.

Example

```
run pl1 source -map -table
```

Causes the PL/I compiler to be invoked in a run unit. Upon return from the run command, no reference names, segment numbers, etc., generated by the pl1 command remain. (Note temporary segments used by pl1 are freed but still remain in the process.)

```
run -use >udd>Proj>Pers>alloc_ pl1 foo
```

causes a different allocation program to be used during the run.

Notes

A QUIT during a run does not cause exit from the run. Any activity performed while the program being run is suspended is forgotten when (if) the run unit is terminated.

A run unit sets up a "condition wall" so that programs before the run unit manager on the stack do not get control until the run is terminated (possibly because of a release).

If the specified time limit is exceeded, the user is asked if he wants to continue with the run. If the response is "yes", another slice of time, the same size as before, is made available; if the response is "no", the run unit is terminated.

When a run unit is terminated, the "finish" condition is signalled to all programs still active on the stack that are part of the run unit.

All I/O switches attached or opened during the run due to language I/O are reset to their prior state. Any action taken by the io_call command or by explicit iox_ calls within a run are not reverted when the run is terminated.

Any temporary segments used by programs during the run should be cleaned up by the same programs. The run unit manager does not attempt to clean up temporary segments.

Name: run_unit_manager_

The run_unit_manager_ subroutine manages the environment for a run unit and invokes the main program of the run.

Entry: run_unit_manager_$environment_info

This entry enables debuggers to obtain the saved stack header information used by a given stack frame.

Usage

```
declare run_unit_manager_$environment_info entry (ptr, ptr,
            fixed bin (35));

call run_unit_manager_$environment_info (stack_frame_ptr,
            info_ptr, code);
```

where:

1. stack_frame_ptr
   points to an active stack frame on the current stack. (Input)

2. info_ptr
   points to the following structure to be filled in:

```
dcl 1 env_ptrs aligned based,
    2 version fixed bin,
    2 pad fixed bin (35),
    2 iot_ptr ptr,
    2 isot_ptr ptr,
    2 clr_ptr ptr,
    2 combined_stat_ptr ptr,
    2 user_free_ptr ptr,
    2 sys_link_info_ptr ptr,
    2 rnt_ptr ptr;
```

   where:

   1. version
      is the version number of this structure; currently it is 1.

   2. pad
      is unused.

3. lot_ptr
   points to the linkage offset table (LOT).

4. lsot_ptr
   points to the internal static offset table (ISOT).

5. clr_ptr
   points to the area where linkage sections are allocated.

6. combined_stat_ptr
   points to the area where separate static sections are allocated.

7. user_free_ptr
   points to the area where user storage is allocated.

8. sys_link_info_ptr
   points to the control structure for external static variables.

9. rnt_ptr
   points to the reference name table.

3. code
   is a standard system status code. (Output)