To: MTB Distribution

From: Melanie Weaver and Charles Hornig

Date: August 12, 1982

Subject: Tasking I

Send comments by one of the following means:

By Multics mail (on System M or MIT):
Weaver.Multics

By Telephone:
HVN 261-9312 or 617-492-9312

By Forum on System M (preferred method):
>udd>m>mbw>mtgs>tasking

INTRODUCTION

This MTB proposes changes to the Multics system to support user ring tasking. Tasks are sub-process entities whose execution can be interleaved. The proposal includes general descriptions of commands and subroutines to manipulate tasks, a scheduling mechanism, and changes to the system to deal with the more complex environment.

It is becoming increasingly urgent to have a fully-supported tasking facility on Multics. Already the ARPA network and emacs use a prototype version that has evolved over several years. Other products slated for the MR10 time frame need it too, such as the expanded mail facility and the proposed inter-Multics forum. It is especially suited to applications that involve servers, making them faster to write, easier to maintain, and more robust. Tasks can also improve the command level interface by isolating different activities. Tasking is also being considered as a means of implementing cheap processes for a proposed Unix subsystem. In the future, we will need to support tasking/real time extensions to PL/I, Ada, Fortran and/or Basic.

The existing prototype is not adequate by itself. It has little documentation and is thus difficult to understand. It creates a more complicated environment which the system has not yet caught up to. Because of this, its use must be restricted. Users are temporarily putting up with glitches that cannot be tolerated in the long run.

The job of documentation, cleaning up the software, and improving system support should take less than a person-year. The tasks to be done are listed in Appendix A. The proposed user-ring version is simpler and cheaper than implementing multi-ring simultaneous tasks and is adequate for most of our near-term needs (next year or two). Most work done on this project will not be wasted in the event that we implement the more general mechanism.

This MTB does not include the new/revised interfaces, commands, include files, etc., although the prototype subroutine interfaces are described in Appendix C. It also does not present plans for implementing tasking in specific languages. Instead it describes the basic mechanism, how it affects the system, and what types of changes will be necessary.

SUMMARY

The proposed mechanism allows a process to have several tasks, each with its own stack. Some tasks will share a LOT, ISOT, user free area and standard I/O switches, while others will have their own versions of these. Some tasks (as part of a run unit) will have their own RNT (reference name space). There may be several task groups, each with its own LOT, etc. The first task in a run unit always gets a new LOT, etc.

The advantage of a task group is efficiency. LOTs, linkage sections, etc. don't need to be reinitialized for each task. Attachments for the standard three switches don't need to be moved when switching to another task in the same group. However, since all of a group's tasks share the same internal static and external variables, all programs used in a task group must be careful in their use of these storage classes.

Arbitrary programs that haven't been "cleared" to run in task groups should be used only in tasks that have their own exclusive environment. Perprocess static programs have their static shared be all tasks. If a program wants to keep static data that is task specific, i.e. not shared by any other task even in the group, it must access the data via an external mechanism, such as value_ with a pertask option. This would include data about the task's stack.

There are both subroutines and commands to manipulate tasks. These include facilities to create, restart, stop and obtain meters for other tasks. Each task has an ID. A task can be created to run a command or a program can be invoked in an existing task.

Locks set by set_lock_ will continue to be perprocess. One task can only lock out another process, not another task.

Scheduling of tasks is not done by arbitrary time limits or quanta. Rather a task runs until it
- goes blocked, or
- suspends itself, or
- gets preempted by an event call, or
- requests that another task be scheduled, or
- reaches a time limit set by its creator, or
- returns from an explicit call, such as an event call channel call.

A quit or other condition that goes to command level will cause the task to be suspended so that the command level "task" can be scheduled. A suspended task will not be rescheduled unless explicitly requested or unless it gets a wakeup (if it was blocked). When the scheduler runs, it picks the first task in the list of tasks waiting to run that has the highest priority.

The current tasking mechanism has evolved over several years and now meets most of the needs of the restricted server environments it is used in. The system, however, does not adequately support several parallel tasks in a process. Most of the changes necessary, including uses of static, are required for any reasonable tasking mechanism.

Most of the changes are related to the dynamic linking mechanism, which maintains much of the environment including LOTs, ISOTs, reference names and linkage sections. With tasking, a process may have several such environments. Most linker actions affect only the environment currently in use. However some actions are process wide in scope and must affect all the environments. The system must be changed to operate in multi-environment mode.

One of these actions is segment termination. This should cause all of the ring's LOTs, linkage sections and RNTs to be cleaned up. This part of segment termination should be done in the user ring. hcs_$terminate_seg/file, which now only clean up the current RNT, should be changed to invoke term_. As a rule, if segment names are to be deleted from the RNT, the linkage sections should be searched for snapped links. There should be new ring 0 interfaces that don't do any reinitialization. Programs should be very careful about terminating segments that aren't about to be deleted or truncated.

Another linker activity that has process scope is setting LOT and ISOT entries of segments with perprocess static. The LOT/ISOTs should be initialized to lot/isot faults so that the linker doesn't have to update them all ot task_create_ have to initialize them separately. However, the lot/isot_fault_handlers must know how to find the values. The linker must maintain a perprocess LOT and ISOT for segments with perprocess static.

The run unit mechanism also must change. It was designed to be executed sequentially, not in a process with parallel tasks. The two mechanisms are incompatible, since changes made by one would not be propagated properly in the other. In addition, the change to initialize the LOT to lot faults, needed for efficient tasking, will invalidate the whole (expensive) algorithm now used by run units to clean up. Run units should be reimplemented using tasking by adding an option for a separate RNT.

At process termination time, all tasks should be terminated in an orderly, graceful way. The last task to be terminated is each task group should call execute_epilogue_. The original task (on stack_4) should close the remaining iocbs. There is an urgent need for a process to unwind its stack (i.e. invoke cleanup handlers) when it terminates. This should be done by each task as well.

The tasking mechanism is only available in the user ring. Even with this restriction it is suitable for use in daemon-style servers, language-defined tasking, co-routines, compiler writing and command level organization. It will significantly increase productivity in these types of applications.

The two diagrams following this section illustrate some of the task structure ideas mentioned above. Figure 1 shows the relationships among six tasks in a process. It shows how user environment data structures are shared by task groups and run units. Figure 2 shows the different states a task can have and which task_ctl_ entries change states. It lists some of the reasons for (re)starting and suspending tasks.

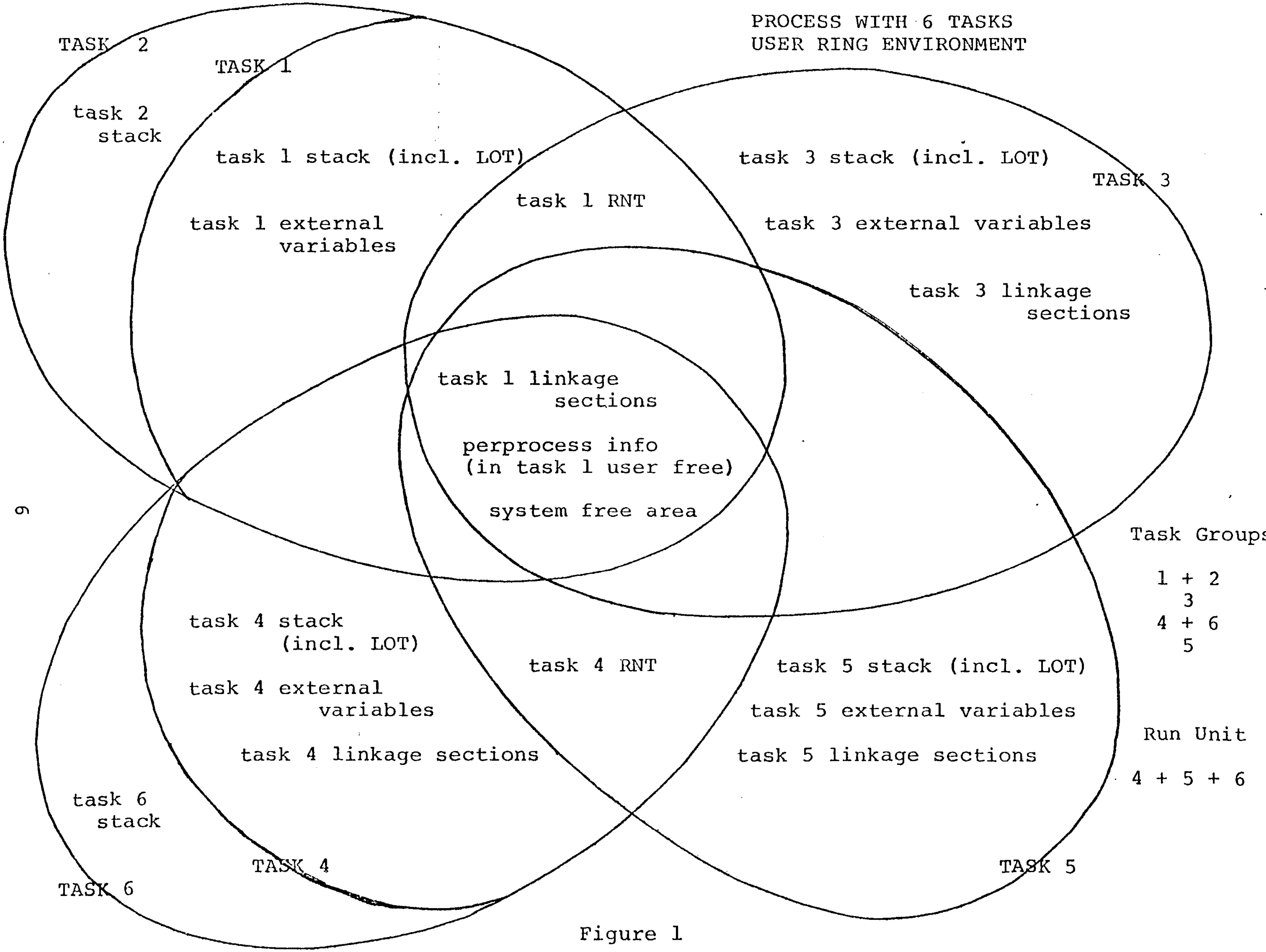
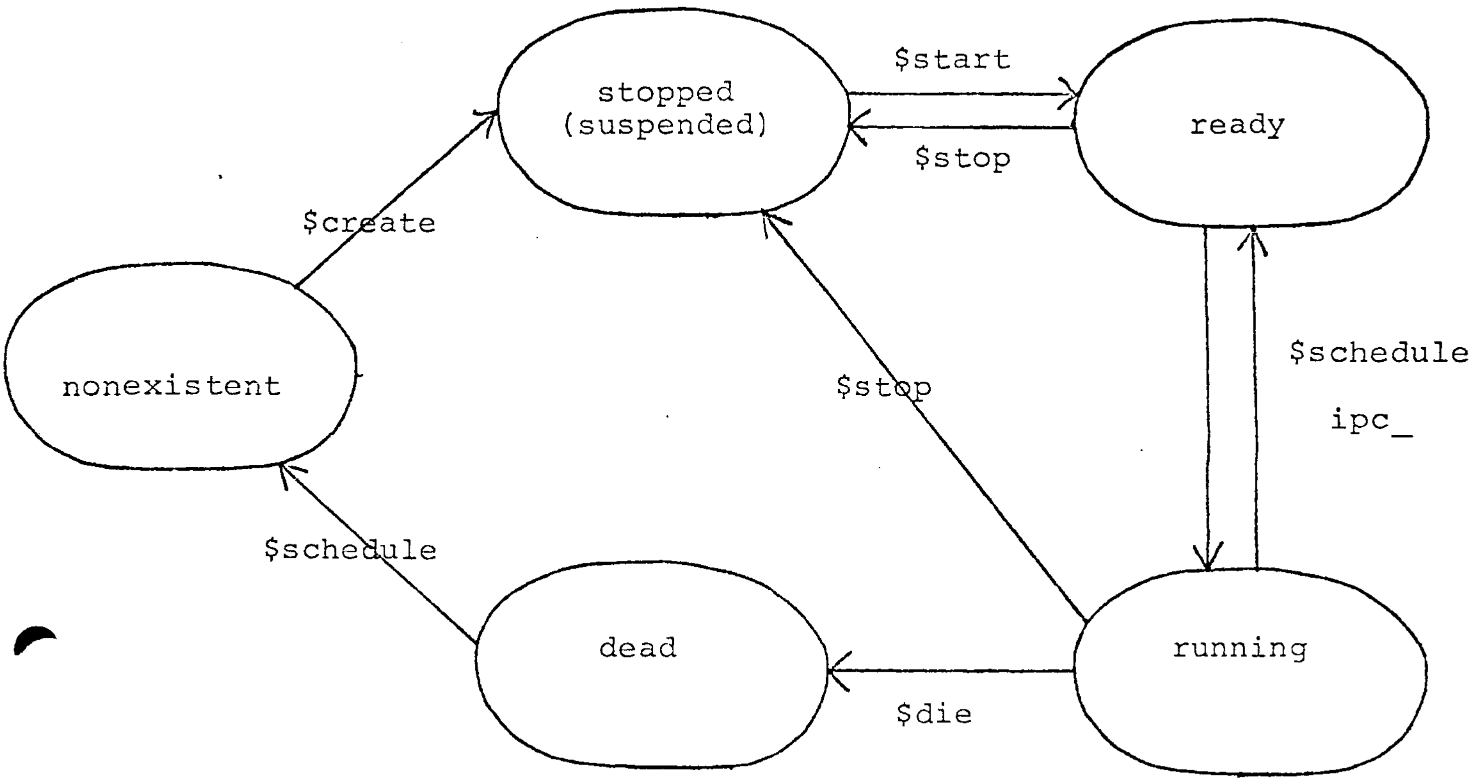


Figure 1

Assumes that linkage sections of perprocess static segments are allocated in task 1's user free area.

TASK STATES

and task_ctl_ entries that change states



$start is used

- after creating the task
- when an ipc_ wakeup has arrived
- after a condition (from command level in another task)

$stop is used

- when the task goes blocked
- when a condition handler tries to get to command level
- when another task wants to be sure the task won't run for awhile

Figure 2

BASIC TASKING MECHANISM

A task is a subset of a process and is identified only in the user ring. Lower rings do not differentiate between tasks since they primarily perform "utilities" on behalf of the user ring. Each task has its own user ring stack. It may either have its own LOT, ISOT and user free area (expensive type) or share those of its creator (cheap type). Tasks that share the same LOT are called a task group. Execution of several tasks may be interleaved.

Each task's stack is a separate segment. These segments always retain their own segment numbers, i.e., an executing task's stack does not necessarily have segment number = ring zero stack segment number + ring number. This would be a problem if tasking were to run in inner rings, since the hardware automatically sets pointer register 6 by the above algorithm when transferring into an inner ring. However the hardware does not set pointer register 6 in the outermost ring used.

Each task has a associated data structure that contains such things as meters, the current state, priority, scheduling information, and threads to sibling and parent task data structures. It is allocated in system free area and accessed through a pointer in the task's stack header.

Tasks are managed by the subroutine task_ctl_ and by ipc_. Many of the task_ctl_ features are described below. The ipc_ features will probably be described in another MTB but are summarized below because they are an important part of the general mechanism.

TASK CREATION

Tasks are created by task_ctl_$create. The information given to task_ctl_$create includes pointers to the task overseer and argument data structure, as well as cpu limit, priority, and environment flags. See the attached documentation for task_ctl_. task_ctl_$create assigns an ID, creates a stack, initializes the stack header and related environment and sets up a stack frame on the new stack for task_alm_, an alm control procedure. It then returns to its caller, leaving the new stack frame with its return pointer set to task_alm_.

The new task is not runnable until task_ctl_$start is called. This entry threads the task into the task scheduler's ready queue. When at last the task is scheduled, it returns to the most recent frame on the stack. Thus the fledgling task awakes in task_alm_. task_alm_ calls a system-supplied task_overseer_, which sets up an any_other(system) handler, moves the standard I/O attachments to per task group iocbs if switching task groups, and calls the task overseer specified in the task create data.

TASK DESTRUCTION

A task is destroyed by getting it to return to the task_overseer_ frame. There are two ways to do this. One is for the user-specified task overseer to return (possibly after a goto). The other is for the task to call task_ctl_$die, which does a nonlocal goto to task_overseer_. The finish condition is not signalled.

Task_overseer_ returns to task_alm_, which calls task_ctl_ to unthread the task from the ready queue, mark it as dead, and invoke the scheduler. The scheduler checks for dead tasks and destroys them, which involves "destroying" stacks, iocbs, etc.

SCHEDULING

Task scheduling is currently performed by task_ctl_$schedule. Each ready task (started and not stopped or killed) has a priority and is in the ready queue. The scheduler picks the first task in the ready queue with the highest priority and then moves the task to the end of the ready queue. Scheduling a different task involves the following actions:

- save the current task's ips mask
- save the standard iox_ switches
- reset the "official" process stack pointer for the ring (stored in the PDS)
- save/restore the current value of PR6 from a location in the stack header
- change cl_intermediary
- restore the standard iox_ switches with the values saved in the new task (when switching task groups)
- restore the new task's ips mask

task_ctl_$schedule returns in the new task. This entry is usually called by ipc_$block, which wakes up another process only when there are no tasks ready to run. Block removes the task from the ready queue and wakeup restores it.

ipc_ - Task Interaction

As mentioned above, ipc_ and the tasking software must cooperate to schedule tasks. In addition, ipc_ creates a separate task for each event call channel. Currently the tasking features are in a separate version of ipc_ and have been merged with the recent ipc_ improvements. This will be discussed in more detail in another MTB.

It has been suggested that tasking primarily use something other than ipc_ since it is all within a single process. However, the new mechanism would have to be similar, and there would still have to be interaction with ipc_ for compatibility. The ipc_ facility of optionally creating separate tasks for event call channels simplifies their use.

The ipc_ mechanism should be able to handle the expected task scheduling requirements. Device management uses call channels. Language-defined calls will mostly use wait channels (perhaps with the argument list pointer passed as the message). There will need to be some extensions, such as the one to handle the wait with time limit feature needed by Ada.

METERS

Process usage meters are kept for each task in the task's control data. They are incremented whenever another task is to be scheduled. The actual scheduling code is charged to overhead meters. There are a subroutine and a command to obtain the values.

TASK - COMMAND LEVEL INTERACTION

Getting to Command Level

Command level is generally in the original task. The process's first stack in the user ring is considered the original task even if tasking has not been explicitly initialized. Whenever another task wants the user to get to command level, such as after a condition, it must stop itself and schedule the original task. To accomplish this, all other tasks establish a special cl_intermediary, which is called when condition handlers come to command level. cl_intermediary suspends the task, which means that it can only be restarted explicitly, and invokes the scheduler. Since the original task has a high priority, it gets scheduled quickly.

Terminal I/O

Each task group has its own standard iox_ attachments which all start out synned to user_i/o. Terminal I/O logically belongs to the original task. There is no indication of which task issued an output line. Of more concern, it is generally impossible to determine which task a piece of terminal input belongs to if more than one task expects input on the same I/O switch (e.g. user_i/o). In the long run, the I/O / video system as a whole should be changed to handle this situation.

For the time being, an adequate solution is for the command that invokes a specified command in another task to temporarily stop the current task. This prevents the caller from executing (and thus doing I/O) while the callee is executing. The assumption is that both tasks do terminal I/O. In general, the calling task in this case is the original/command level task which must be restarted when needed. If the called task signals a condition and the handler tries to come to command level, that task's cl_intermediary restarts the original task.

Another alternative may be for each task group that is to do terminal I/O to have its own window. This can be specified at task creation time or dynamically. Most tasks will not be doing terminal I/O. At this stage of the video system development, there is not likely to be any identification of the windows' owners or of which window, if any, is currently active. There are still several problems to be solved with this approach.

Ideally there should be an integrated task/video desk management system.

Task Commands

At command level, there are commands to manipulate other tasks. In particular, the following facilities are available:

list	display the tasks' status and numeric identifier
abort	cause the specified task to be cleaned up and destroyed
stop	cause the specified task to be suspended
start	cause the specified task to be resumed
execute	cause the specified task to execute the specified command line

By using the execute facility, for example, one can cause probe to be invoked in a task that was stopped by a condition. Commands for the above exist but need to be refined before being installed.

ARGUMENT PASSING

There are two explicit ways that arguments can be passed to another task--via task creation and via the generate_call facility (which invokes a program in another task). They can also be passed in subroutine calls to other tasks in PL/I and Ada tasking. Tasking imposes some restrictions on the passing of arguments. The argument list and the arguments themselves must exist for the lifetime of the program invocation in the called task, which may be different from the lifetime of the caller. The safest place for them is thus in system_free_area, which is perprocess, although they could be copied into some other shared segment or directly into the target task.

It is expected that the interfaces included in the appendix will change, most likely to include an argument list pointer. The tasking subsystem should be changed to free the argument list allocation as soon as the task or program invocation ends. To make these interfaces more useable, there should be a program to copy arguments into system_free_area and create an argument list there. The interface will be described elsewhere.

CO-ROUTINES

Tasking should include a co-routine facility. Co-routines are necessary for simulation languages such as Simula and are recommended for improving the organization of compilers. They significantly reduce compiler complexity. The current PL/I compiler often uses internal static variables to simulate co-routines.

Co-routines are implemented on separate stacks but have different scheduling requirements from ordinary tasks. The switching overhead must be about the same as a single procedure call. However, the requirements are simpler than for general tasking. Co-routines are synchronous; two interacting co-routines cannot both be active at once. The target is known during "calls" and "returns", so the priority mechanism can be bypassed. The current ipc_ entries cannot be used in this case, but it should not be difficult to add the necessary capability to task_ctl_ (or possibly ipc_).

LOCKING

Locks set by set_lock_ will continue to be perprocess. This is consistent with inner rings having no knowledge of tasking. set_lock_ cannot wait in an inner ring for a lock set by another task in the same process, since the other task cannot run until the inner ring returns. Thus tasks cannot use set_lock_ to lock out other tasks. Locks can still be used in inner rings, but waiting for them will block the whole process.

Eventually it will be desireable to have per task locking. In particular, the Data Management System (DMS) would then be able to handle multiple simultaneous transactions in a process, with each in a separate task. Since part of the DMS is to run in ring 2, there will have to be a way to deal with busy locks in inner rings. There are two general approaches to handling an inner ring's discovery of a lock locked by another task.

One approach is to continue restraining the task mechanism to the outer ring. In this case, an inner ring that tried to lock a lock locked by another task in the same process would have to back out of all its work and return to the outer ring. The task would then have to go blocked to allow the other task to run. This is unacceptable because it is too clumsy and expensive.

The other approach is to extend the tasking mechanism to lower rings, at least to ring 2 and preferably to ring 0. This would cause a process to potentially have several stacks in each ring. In this case, a program such as set_lock_ could go blocked when it encounters a lock locked by another task. The task scheduler could then resume another task in whatever ring that task was stopped. This approach has its own significant implementation issues which should be discussed in another MTB.

RUN UNITS AND SEGMENT TERMINATION

The current run unit mechanism will not work in conjunction with tasking. It was designed to work with only one stack in the user ring. It depends on the fact that execution of a run unit and its parent environment are not interleaved. When a run unit terminates, the run unit manager attempts to clean up the environment heuristically by comparing LOT entries. Based on this information, it updates perprocess static and terminates segments. It also unsnaps all links in perprocess static segments that were snapped during the run unit.

Besides being expensive and heuristic, this mechanism will not work in a tasking environment. Other tasks are executed in parallel, possibly using some of the same segments, including perprocess static ones. The environment must be kept up to date. Also the LOT comparison method itself will become impossible. It depends on uninitiated segments having LOT entries of zero and initiated segments with no active linkage sections having LOT entries set to lot faults. As explained below, there are two reasons why LOT entries should always be initialized to lot fault.

While invalidating the current mechanism, tasking itself can replace it. It already provides many of the same features. Two that it does not currently provide are the option of a separate RNT (reference name table) and automatic termination of segments used only by the task/run unit. This section discusses the changes needed to support run units under tasking. The resulting new mechanism will be simpler and more robust.

Segment termination is an issue even if run units are ignored. In this case also, the current mechanisms assume that there is only one environment to clean up. It is discussed in this section because it is related to and complicated by the changes for run units.

RNT and LOT Reinitialization

The main change needed is an option in task creation to create a new RNT for the task. The separate RNT feature of run units is used in the field and should not be eliminated from the system.

Adding and deleting the RNT itself is not difficult. However, multiple RNTs are a problem when terminating a segment. In that case, the reference names for that segment should be deleted from all RNTs in a ring. This is not always done in run units today. Currently, segment termination and reference name management are both done in ring 0, even though the RNTs are not in ring 0.

This is related to the non run unit problem of how to reinitialize all of a segment's LOT entries when the segment is terminated, also currently done in ring 0. makeunknown_ sets the segment's current LOT entry to zero. As explained below, we would like to make the default LOT entry value always be a lot fault (currently lot faults are set by initiate). Then a LOT entry will need to be reinitialized only if there is an active linkage section, which can be done by term_.

Given the above change, hcs_$terminate_noname is not likely to cause trouble because it only deletes one null reference name. It will not terminate a segment that has an active linkage section or a name in any RNT (unless someone uses hcs_$terminate_name incorrectly).

The issue is not simply extending the mechanism in use today. Currently most programs call term_ to terminate a segment that might be the target of snapped links and/or have an active linkage section. It runs in the user ring. hcs_$terminate_seg/file are called for other segments and only clean up the RNT and the LOT. However one task cannot so easily assume that a segment is not being used by another task. There are exceptions, such as "private" segments of programs that are not likely to run in more than one task. But heavy-handed terminating, even with proper cleanup, should be reserved for segment deletion or truncation. Interfaces that don't thoroughly clean up should be used with great caution.

Keeping this in mind, the problem at hand is to upgrade the current interfaces to be more robust under tasking. term_ must know how to clean up multiple LOTs, linkage sections and RNTs. It should call new hcs_ entries that terminate a segment without reinitializing the environment. Some of this work has already been completed.

We feel that most callers of hcs_$terminate_seg/file should be calling term_ instead. A new entry in term_ could provide the optimization of searching linkage sections only if there are non-null reference names and/or active LOT entries.

It would probably be too incompatible to force all the callers of hcs_$terminate_seg/file to be changed abruptly, for example by deleting the entries or making them no longer work. It would be better to have hcs_$terminate_seg/file somehow invoke term_.

There are at least two ways to do this. One is to move term_ to ring 0 (bound_sss_active_). That would force ring 0 to know about multiple LOTs, RNTs, etc. The job of cleaning up the user ring environment belongs in the user ring. The other way is to effectively make hcs_$terminate_seg/file be writearounds to term_ in the user ring. This can be accomplished by changing hcs_ and the linker to provide a kind of automatic resolve_linkage_error facility. The entries to be deleted would be added to a table along with their replacements. When the linker detects an external symbol not found error, it searches the table for a replacement before returning a linkage error. Then some of the hcs_ entries could be routed to term_.

In any case, hcs_$terminate_seg/file cannot remain as they are today, updating only the LOT and the RNT of the current task. At least they have to remove reference names from all RNTs and make sure that all LOTs contain lot faults for the segment.

If ring 0 is to continue to reinitialize LOT entries and to delete reference names, it must have available lists of all the LOTs and RNTs. The lists would be accessed through a perprocess information structure. Some such structure is needed anyway for handling perprocess static--see below. The lists would have to be kept up to date, but we feel that this is preferable to having ring 0 know about the format of the actual task control data.

Automatic Segment Termination

The other run unit feature to be discussed is the automatic termination of segments used only by the task/run unit. This is expensive, somewhat heuristic, and depends on the way LOTs are initialized. Currently a segment is automatically terminated if its entry in the non-run unit LOT is zero and if it is not perprocess static or a temp segment or part of a known area. However in tasking it becomes much harder and more time-comsuming to figure all of that out.

This method of finding the segments would also make initiate more expensive under tasking. It depends on having LOT entries of zero for unused segment numbers. Currently initiate fills in lot faults, so it would have to be changed to find and update all the LOTs. It is necessary for all LOTs to contain lot faults for all initiated segments, because a segment can be executed in a task without having been referenced through the linker and having its linkage section combined. It would be more efficient to simply initialize the LOTs with all lot faults.

Actually automatic termination of segments could be eliminated. It does not affect the functionality of run units. At worst there would eventually be several initiated segments that nobody knew about. At least, nothing would get terminated by mistake.

There is another way to do it which should have been done in the first place. It depends on keeping the reference name count up to date. Currently run units create and delete RNTs without updating the reference name count.

First we must make sure that the reference name count is correct at the beginning of the run unit. The copy RNT option should add the number of names in the new (copied) RNT to the reference name counts in ring 0.

When deleting the RNT at the end, the run unit manager should decrement the reference name counts by the number of names in the RNT. There is no need to delete individual RNT entries in this case. Instead, a new hcs_ entry should decrement the reference name count and terminate the segment if the count goes to zero.

This method will not terminate any segment that has null reference names. Individual programs are still responsible for terminating those. This will automatically prevent segments such as temp segments and area components from being terminated. Any segment used outside the run unit will have null reference names and/or names in another RNT. Since run units are intended only for fairly self-contained programs, it is reasonable to assume that a run unit will not pass pointers to tasks outside the run unit. In other words, it is unlikely that tasks outside the run unit will use the run unit's segments while bypassing the reference name count.

Run Units' Spawning of Tasks

(The reader should know that run units come in three varieties: -old_reference_names, -copy_reference_names and -new_reference_names.)

Run units may spawn other tasks and run units. Except for run units with their own RNT, these should share the RNT of the creating run unit (rather than the RNT of the original task). Spawned tasks should be able to have their own LOTs, since this feature will be needed by PL/I tasking.

Generally all spawned tasks should be terminated when a run unit ends. (See below for a discussion of task termination.) At least an RNT cannot be deleted while there are any tasks left using it. There are a couple of ways to find the tasks to be terminated.

One is to wait until an RNT is about to be deleted (by the run unit that created it) and then terminate all tasks using that RNT. This will leave intact all spawned run units that have their own RNTs. Tasks spawned by a run unit that uses the original RNT will not be terminated until process termination.

Another scheme is to assign run unit IDs which would be propagated to all spawned tasks. All spawned run units would be threaded together. Before a run unit is terminated, all of its spawned tasks and run units (except possibly run units with their own RNTs) could be found and terminated.

This could be simplified somewhat by not allowing run units to spawn other run units. A process could still have several run unit tasks at the same time.

LINKAGE AND STATIC SECTION CHANGES

Perprocess Static

Tasking will not work properly until the system provides more robust support of perprocess static segments. After several years of experience with run units and the prototype tasking, we have found most of them (see appendix B for a list of them). The problem is that a perprocess static segment may first be used in one task and then used again in another task that does not share the same LOT and ISOT. The second task, however, must use the same copy of the segment's static section even if it did not exist when the segment was first referenced. Current run units do not have this problem because the run unit manager sees to it that perprocess static segments' linkage and static sections are properly updated/cleaned up when a run unit returns. This is clearly not feasible for tasks, which run in parallel.

A solution is to have a process ISOT (PISOT) which the linker checks before combining the static section. A complete solution is more complicated because a discussion of static sections cannot be separated from a discussion of linkage sections. It also turns out that what happens to iox_ is a major consideration. There are several alternatives for handling perprocess segments' static and linkage sections, each with different side-effects. First we present the one we prefer, followed by others to allow the reader to make a more informed judgement.

ALTERNATIVE 1

Continue to have all tasks share the linkage sections of perprocess static segments. The main reason this is currently done is that no perprocess static segments have separate static--it is always part of the linkage section. This has the following implications:

- There must be a PLOT and a perprocess set of external variables as well as a PISOT. The linker would always allocate perprocess linkage and static sections in the original task's linakge area.

- Perprocess static segments must always use the same RNT, since all tasks share the links. This may mean more complicated RNT management. Changes to specify which RNT to use are much simpler than having a single RNT with different "branches" for each run unit. The current run unit mechanism unsnaps links in perprocess static segments that were snapped during run units. This is not feasible in tasking.

- Likewise, perprocess static segments cannot link to the static sections of non-perprocess static segments or to non-perprocess external variables. This can be enforced by the linker.

- Part of lot_fault_handler_ can be in alm and just copy PLOT and PISOT entries when appropriate. This avoids having to special-case the LOT entry of lot_fault_handler_ (except in the original task).

- Whenever a task from a different task group is scheduled, the user_input, user_output and error output switch attachments must be moved. (The actual switch is in iox's static.) This means that a switch synned to user_output always uses the attachment of the task it is used in, rather than that of the task it is defined in.

ALTERNATIVE 2

Have only static sections be perprocess. All linkage sections would be per task group. iox_ would no longer be perprocess static. Implications of this alternative are:

- Since each task has its own standard switches, it is no longer necessary to do move attaches when switching to a different task group.

- iox_ and print_attach_table must be able to deal with several switches named user_input, user_output and error_output.

- iox_'s perprocess information, e.g. user_i/o and the iocb name space, are initialized when a task is created. The iocbs themselves are allocated in a perprocess area.

- The user_output, etc. switch names will always map into the task's local switches. A switch synned to user_output always uses the actual attachment of the task that made the syn attachment.

- All perprocess static must be separate from linkage sections. The error table macros must be changed to optionally generate separate static. (The binder cannot internally resolve links to other components' separate static because the static pointer is not in a dedicated register when the reference is made.)

- There need be no PLOT.

- The local RNT is always used.

- lot_fault_handler_'s LOT entry must be filled in when the task is created, so its linkage section must be either shared or pre-combined.

OTHER ALTERNATIVES

Same as alternative 2 but have a separate perprocess static object segment section for iox_ instead of initializing perprocess information when tasks are created. This would only be generated by alm and the binder, since iox_ is probably the only segment that needs both types of static. Having to support a whole extra object segment section seems an excessive cost for not having to do move_attaches.

Combine alternatives 1 and 2 by forcing all perprocess static to be separate but keeping iox_ all perprocess. This avoids the iox_ confusion but has the overhead of both the extra separate static and the move_attaches.

Combine alternatives 1 and 2 by sharing linkage sections and reimplementing the way iox_ handles the three standard switches (possibly with builtin functions). This would create even more iox_ confusion and is probably too incompatible.

Pertask Static

Distinct from the above discussion, some programs need per task (not task group) static, for example to keep data about the stack they are running on. These include cu_$get/set_cl_intermediary, trace and probe. A new pertask object segment section would not be adequate. It cannot be linked to, because most links are per task group. probe and trace each have several modules that must access the data, so the task data must be external. A solution is to have something like a per task value segment, perhaps adding a pertask option to value_. This can be used by user programs also. Stack header variables would work but are less flexible and limited to a few system programs.

EXTERNAL VARIABLES

External variables are used to implement the storage classes of PL/I external static and Fortran common (when not specified to be in the Multics hierarchy). They are referenced through *system links. The linker allocates them as a threaded list in the user free area. The list is accessed through a pointer in the stack header.

There must be a separate set of these variables for each task group in order to adhere to PL/I tasking rules. This makes their use by perprocess static programs confusing. It is unacceptable for these programs to reference different variables in different tasks. Therefore perprocess static segments must use perprocess external variables. There are at least three ways to do this:

- Have perprocess static segments use only the external variables of the original task. This may be best if that task's RNT is to be used by perprocess static segments.

- Implement a separate, perprocess set of external variables.

- Prohibit perprocess static programs from using external static. This can be enforced by the linker. Change the few that do to use perprocess value_ variables or perprocess static cds segments instead.

REALLOCATING LOTS

A LOT that is currently being used must contain a valid entry for every initiated segment. Sometimes the LOT fills up and must be enlarged, which always involves reallocating and moving it. This affects every task, since each has a lot_ptr in its stack header. Each task group's LOT must be grown. All tasks in a task group must have their lot_ptrs changed to point to the new LOT.

The first LOT enlargement is usually done in ring 0 by initiate. The problem is how to get all the other LOTs enlarged and update all the lot pointers. We do not want ring 0 to have to know how to find all the task groups.

All the updating can be done in the user ring by the task scheduler. Before running the new task, the scheduler can check that the LOT sizes are the same. If the new task's LOT size is smaller, the scheduler will change lot_ptr to point to the new LOT. If the new task is in a different task group, and if that group's LOT has not yet been grown, the scheduler will grow it.

RNT ALLOCATION

Tasking will force a change in RNT management, even if there is no multiple RNT feature. Currently the RNT consists of a header and a threaded list of names. Both are allocated in the RNT area to minimize page faults. If the area fills up, a larger one is created and the old one copied into it. This is done in ring 0 and includes changing the rnt_ptr in the stack header. The problem in tasking is that there are several stacks, each with a copy of the rnt_ptr. If nothing is changed to deal with this, reallocation will cause most of the rnt_ptrs to become invalid.

There are several possible solutions:

- Teach ring 0 to update all stack headers that contain the same rnt_ptr.

- Start with a larger RNT area and don't reallocate it.

- Don't have a separate RNT area. Use system free area instead. This may cause more page faults.

- Remove the RNT header and search rules from the RNT area so that they will not be reallocated. If the RNT area gets reallocated, there may be more page faults. This is the alternative we prefer.

PROCESS / TASK TERMINATION

During process termination, all tasks must be terminated in an orderly fashion. The environment must be properly cleaned up, which is not done today. This section summarizes the proposed mechanism, then discusses some details.

All normal process termination (when the environment is intact) will be funneled through the common code in logout. terminate_process_ should be used only by logout, etc. or when the environment is too sick to clean up (e.g. when there are bad iocb threads). logout will cause each task to terminate itself by signalling finish, unwinding the stack, and, for the last task in a task group, calling execute_epilogue_. The original task will be the last. After it has finished the usual termination, it will close iocbs, call execute_epilogue_ in inner rings, and call terminate_process_.

The common code in logout will be changed to call task_ctl_$terminate_process. This entry will set up invocations of the terminate_task command in all but the original task. It will rearrange the task priorities so that the scheduling will be done in the correct order, with the original task last. If logout was not invoked in the original task, an invocation of another command entry in logout will be prepared in the original task with arguments specifying the reason for process termination, absentee logout message if any, etc. Before scheduling any other task, task_ctl_ will disable its priority setting and task creation features.

terminate_task will signal finish and cleans up the stack. This will be done }ieither by a nonlocal goto to task_overseer_ or by calling unwind_stack_. The latter offers more control. The caller can set up a time limit and an any_other handler. Also in this case at least, if not in the general case, cleanup handlers should be prevented from doing nonlocal gotos and thus circumventing the whole operation. If terminating the last task in a task group (probably known by an argument), it will call execute_epilogue_. The code currently in execute_epilogue_ that closes iocbs will be moved to another procedure. When done, terminate_task (or task_overseer_) will call task_ctl_$kill, which sets the task's dead bit. When the scheduler is next invoked, it checks for dead tasks and destroys them.

Back in the original task, logout will signal finish, call unwind_stack_, call execute_epilogue_, print the logout message, close all iocbs and call terminate_process_. If a process is not using tasking, calling task_ctl_$terminate_process_ has no effect

It may be possible to speed all of this up somewhat by only signalling finish within run units. This requires changes to the

PL/I manuals and assumes that system programs do not have specific finish handlers.

APPENDIX A

Remaining Work

Define and document new system/user structures and interfaces.

Clean up tasking software, including the commands.

Add options to task_ctl_$create to make a new RNT.

Change the linker to maintain lists of perprocess static segments and to properly allocate their linkage and static sections.

Either change ref_name_ and/or callers to know about multiple RNTs or make all perprocess static segments have separate static.

Add automatic segment termination facility for run units based on reference name counts.

Change lot_fault_handler_ to handle perprocess static segments differently.

Finish changing term_ to update all LOTs, RNTs and linkage sections.

Add new hcs_ entries for segment termination.

Change hcs_$terminate_seg, etc, to reinitialize LOTs, RNTs correctly?

Reimplement run.

Change process/task termination to properly clean up.

Create a mechanism such as per task value or add pertask option to value_. Make sure value_'s perprocess option and its users work correctly.

Change probe, trace, binder and cu_$get/set_cl_intermediary to be per task.

Write program to allocate and copy argument lists.

Coordinate with development of video system for effective desk management system. (longer range?)

APPENDIX B

List of Perprocess Static Segments

bound_audit_
bound_debug_
bound_mail_system_
bound_memo_
bound_metering_
bound_msg_facility_
bound_search_facility_
bound_trace_
bound_command_env_
bound_ssu_
bound_exec_com_
bound_probe_
bound_full_cp_
bound_io_commands_

operator_pointers_
trace_operator_pointers_
bound_ipc_
bound_command_loop_
bound_sss_active_
bound_sss_wired_

bound_extended_mail_
bound_graphics_system_
bound_tp_runtime_

bound_old_cp_

APPENDIX C

Prototype task_ctl_ Interfaces

06/30/82 task_ctl_

Function: Manage multiple tasks within a process. Each task has its own stack, and may also have its own static storage. Execution of several tasks may be interleaved.

Entry points in task_ctl_:

:Entry:create: 12/03/81 task_ctl_$create

Syntax:
```
call task_ctl_$create (task_create_data_ptr, task_id, code);
dcl task_ctl_$create entry (ptr, fixed bin (35), fixed bin (35));
```

Function: Creates a new task and returns its task ID. The task will be in the stopped state.

Arguments:
task_create_data_ptr
   points to a task_create_data structure. (Input)
task_id
   is set to the task ID of the created task. (Output)
code
   is a standard Multics error code. (Output)

Notes: The task_create_data structure is declared in task_create_data.incl.pl1. It contains the following information:

version fixed bin
   version of the suructure,
overseer variable entry (ptr)
   first procedure to be called in the new task,
data_ptr ptr
   pointer to be passed to the overseer,
vcpu_limit fixed bin (71)
   CPU time limit for task (0 if none),
priority fixed bin
   priority of task,
comment char (64)
   description of task for the curious,
top_level bit (1) unal,
   ON if the task is to be independent of the creating task,
shared_static bit (1) unal,
   ON if the task is to share the static of its creator.

:Entry:current_task: 12/03/81 task_ctl_$current_task

Syntax:
```
task_id = task_ctl_$current_task ();
dcl task_ctl_$current_task entry () returns (fixed bin (35));
```

Function: returns the ID of the running task.

Arguments:
task_id
   is set to the task ID of the running task. (Output)

:Entry:die: 06/30/82 task_ctl_$die

Syntax:
```
call task_ctl_$die;
dcl task_ctl_$die entry;
```

Function: Causes the current task to be aborted. The stack will be unwound to its base, and the task will then be destroyed.

:Entry:generate_call: 06/14/82 task_ctl_$generate_call

Syntax:
```
call task_ctl_$generate_call (task_id, procedure, data_ptr, code);
dcl task_ctl_$generate_call entry (fixed bin (35), entry, ptr,
    fixed bin (35));
```

Function: Call a specified procedure within a task.

Arguments:
task_id
   is the task ID of the task in which the procedure is to be called. (Input)
procedure
   is the procedure to be called withing the task. It must be an external entrypoint. (Input)
data_ptr
   is a pointer which will be passed to the procedure. If it is null, the procedure will be called without arguments. (Input)
code
   is a standard Multics error code. (Output)

:Entry:get_task_usage: 12/03/81 task_ctl_$get_task_usage

Syntax:
```
call task_ctl_$get_task_usage (task_id, info_ptr, code);
dcl task_ctl_$get_task_usage entry (fixed bin (35), ptr,
    fixed bin (35));
```

Function: Return usage figures for the specified task. The interface is similar to that of hcs_$get_process_usage.

Arguments:
task_id
   is the ID of the task for which resource usage figures are desired. (Input)
info_ptr
   points to the process_usage structure used by hcs_$get_process_usage. (Input)
code
   is a standard Multics error code. (Output)

:Entry:schedule: 06/14/82 task_ctl_$schedule

Syntax:
```
call task_ctl_$schedule ();
dcl task_ctl_$schedule entry ();
```

Function: Find the highest priority runnable task and dispatch it. If this is not the current task, the current task will be suspended.

:Entry:start: 12/03/81 task_ctl_$start

Syntax:
```
call task_ctl_$start (task_id, code);
dcl task_ctl_$start entry (fixed bin (35), fixed bin (35));
```

Function: Start the specified task. The task will now be considered runnable.

Arguments:
task_id
   is the task ID of the task to be started. (Input)
code
   is a standard Multics error code. (Output)

:Entry:stop: 12/03/81 task_ctl_$stop

Syntax:
```
call task_ctl_$stop (task_id, code);
dcl task_ctl_$stop entry (fixed bin (35), fixed bin (35));
```

Function: Stop the specified task. The task will no longer be consider runnable.

Arguments:
task_id
   is the task ID of the task to be stopped. (Output)
code
   is a standard Multics error code. (Output)