To: MTB Distribution

From: T. Casey

Date: 8 March 1978

Subject: Improvements to Control of Absentee Jobs

INTRODUCTION

This MTB describes enhancements to the absentee facility that allow the system, the central operator, Remote Job Entry (RJE) station operators, and ordinary users, to exercise better control over which absentee jobs are run, and the order in which they are run.

These enhancements have been requested by a number of customers. The need for them will be felt most strongly by sites that use the RJE facility.

The contents of this MTB are an attempt to provide a coherent design for a large set of enhancements that will be implemented in stages over the next several Multics releases.

DEFICIENCIES OF CURRENT FACILITIES

In the current system, the operator can cancel an absentee job only after it starts running, while the user who submitted it can cancel it only before it starts running. The operator has no control over the order in which jobs are run (except for the ability to "turn off" one or more of the lower queues, preventing any jobs from those queues from running, thus reserving the available absentee slots for any jobs that are subsequently submitted in higher queues). The user can only change the order in which his jobs are run by canceling them and resubmitting them in a different order or into different queues.

The system automatically controls interactive logins to prevent one user or a group of users from monopolizing the resources of the system, by limiting the number of logins from each load control group and limiting each user to one process (except for users with the multip attribute). There are no analogous controls on absentee logins.

Further, although the system claims to provide quick service to high priority jobs, by means of multiple queues, it is possible for lower queue jobs to tie up all the absentee slots, preventing the login of higher queue jobs submitted after the lower queue jobs started. Also, because of the limited flexibility of absentee load control (abs_maxu and abs_maxq are settable to constants on each shift), jobs that should be run are sometimes postponed. Setting abs_maxu to a conservatively low value (2 or 3) can prevent high priority jobs from being run as soon as they should, and can prevent low priority jobs from filling idle time as completely as they should.

There is currently no way to synchronize the login of an absentee job that needs some resource with the availability of that resource. For example, an absentee job that needs a tape drive can log in and wait indefinitely if none is available, tying up an absentee slot and inconveniencing the user (who would probably prefer that the job not start if it was going to be unable to finish).

SUMMARY OF THE RJE FACILITY

To understand the reasons behind some of the proposals that follow, it is necessary to know certain things about the RJE facility.

The process that runs an RJE station has a special process overseer that accepts dials from RJE stations, responds to RJE commands, and submits absentee jobs on behalf of RJE users. We will use the term "RJE daemon" to refer to such a process. Although there is no requirement that an RJE daemon be on the SysDaemon project, or even that it be a daemon (i.e., be logged in via the message coordinator), it is expected that RJE daemons will be logged in via the message coordinator (rather than from a terminal at the remote site). The RJE process overseer was designed with this in mind, in that it accepts RJE control commands from both user_input (the terminal or the message coordinator) and slave_input (the card reader or other input device of the RJE station).

The RJE daemon must have the dialok attribute so it can accept dials, and it must have "e" access to the segment >sc1>proxy>absentee_proxy.acs so it can submit proxy absentee jobs (i.e., jobs to be run on behalf of, and under the User_id of, some other registered user).

In general, a given RJE daemon can accept dials from any number of different RJE stations. However, an RJE daemon can have only one station at a time dialed to it.

It is assumed that a given RJE station will always be run by a process of the same User_id - that is, that a given station will always dial the same RJE daemon or else all the RJE daemons will have the same User_id.

There will be a need to identify the jobs from a particular station, for obvious reasons. The User_id of the daemon will not suffice since a given daemon can serve more than one station.

Further, a means of listing which RJE stations are currently being served, and by which daemons, will be useful.

Two other points should be noted. First, from the viewpoint of the system - i.e., the answering service and the message segment facility - the RJE daemons are much like ordinary users, and the absentee jobs they submit are like those submitted by ordinary users, except that they are proxy jobs. The RJE daemons have no special privileges except the dialok attribute and access to absentee_proxy.acs.

Second, since the RJE daemons will use the same facilities as do ordinary users to submit and control absentee jobs, any enhancements to these facilities that are made to support RJE will automatically be available to ordinary users (unless we go to some trouble to build in restrictions, which we do not plan to do).

NEEDED IMPROVEMENTS

The following list is presented to summarize the extensions that customers have requested, and to motivate the discussions that follow. The items in this list should not be interpreted as indications of the final form that the extensions, and the interfaces to them, will take.

The central operator should be able to:

1. Put any job in a hold state (don't run it until told to by operator), and, of course, take it out of the hold state.
2. Run any job, or set of jobs, immediately.
3. Change the order in which any set of jobs will be run.
4. Cancel any job, whether it has started or not.
5. Set a cpu limit parameter so that any jobs with higher cpu limits are automatically held until the limit is raised. (There will be a site settable default value for this parameter, for each shift, analogous to the current abs_maxu and abs_maxq parameters.)
6. Put a running absentee job in the suspended state (process goes blocked and runs only to copy wakeups out of the ITT into its ECT) so that a high priority job can be started at once, without overloading the system, even though all the absentee slots are in use; also remove a job from this state.
7. List the absentee jobs - either all, or a subset selected by specified criteria, and list them in a specified order. For example: list all jobs from a specified RJE station, print

total number of jobs with cpu time limits under 2 minutes, list next 10 jobs in the order in which they will be run, list all jobs in order by cpu limit, etc.

RJE station operators (and ordinary users) should be able to:

1. Re-order their own jobs without canceling and resubmitting them, and without "losing their places" in the queues.

2. Cancel their own jobs even if they have started running.

3. Specify at submission time that a job needs certain resources and should not be started until those resources are available.

4. List their own jobs, with the same options that the central operator has for listing all jobs; in addition, be able to determine the state of their own jobs, as regards their positions in the queues, whether or not they are held (and for what reasons), and whether or not a job is running.

The absentee facility should be able to:

1. Recognize and respond to the requests from operators and users that will support the above features.

2. Enforce sharing of absentee slots among users and among queues, using parameters settable by each site - for example, upper limits on the number of simultaneously running jobs from one user or from each queue.

3. Allow jobs to be deferred until a specified time interval before shutdown. This would be useful to system maintainers, for example to run jobs that do accounting or hardware error reporting. This would require that the absentee facility be notified each time the scheduled shutdown time is changed by the down command.

4. Allow privileged jobs to be guaranteed to be logged in immediately, or at a scheduled time. This is needed to provide a way of creating multiple processes for transaction processing, without requiring operator intervention or communications access for the user who is turning on the transaction processor.

DESIGN ISSUES

The design of these extensions to the absentee facility must deal satisfactorily with the following issues:

1. The implementation of these extensions must be coordinated with Multics releases and with the completion of other projects. Therefore, some desirable features might have to be

eliminated from the design to allow for timely completion, and the next Multics release will contain only a subset of the features that are left in the design.

2. Many of the queue control extensions would be equally useful if applied to the I/O daemon queues. The operator and user commands to control absentee and I/O daemon queues must be designed at the same time to maintain consistency in the interfaces. The same keywords or control arguments should be used to specify the same actions on each set of queues (hold, release, cancel, etc.) and the existing interfaces to each, where certain words are already used, must be considered in designing extensions to the other.

3. It turns out to be very difficult to provide users with additional information about, and control over, their own jobs, without providing them, at the same time, with some information about the jobs of others, thus introducing write down paths. It is the consensus of opinion that the introduction of (even narrow bandwidth) write down paths is undesirable. Therefore these extensions are being designed subject to the constraint that they introduce no new write down paths. This will force some features to be eliminated, or put off until there is time to effect a more secure, but also more costly, implementation.

4. Although extensions to operator commands make up a large part of this proposal, our goal should be to design a system that will do the right thing automatically in most cases, without requiring continual operator intervention. Since "the right thing" will often be different at each site, this means providing site settable default values, per shift or per queue or both, as may seem appropriate, for each of the parameters that the operator commands can set.

5. Several properties of the ring 1 message segment facility (which is used to maintain the absentee queues) should be mentioned here, since they cause some implementation difficulties that have affected the design of the extensions. Message segments are designed to be FIFO queues. New messages can only be added to the ends of the queues. Each message has a unique id, assigned by the ring 1 software. It is a function of the clock reading at the time the message is added. Thus, messages are ordered in the queue by increasing unique id values. Much of the message segment software depends on this, and changing this dependence would be extremely difficult. Thus, any design that supports reordering of absentee jobs must do so in a way that does not change the unique id order of messages in the queues. The two possibilities are to maintain a list, in a separate segment, of jobs in the order they are to be run, or to implement message-reordering primitives that re-thread messages and then exchange their unique ids, thus preserving their order.

There are primitives to rewrite existing messages. However they have two shortcomings, namely that they rewrite messages in place (and thus cannot replace a short message by a longer one), and they change the sender userid from its current value to that of the process doing the rewriting, thus destroying the information about who was the original sender of the message.

6. Because of the extensions to both user and operator commands that control absentee jobs, a convenient means of referring to a particular absentee job is needed. Currently, users can refer to their own jobs only by the pathname of the absin segment when using cancel_abs_request (the only command that has any need to refer to submitted absentee jobs). If a user has several jobs with the same absin segment (for example, translator_absin.absin), it is currently impossible for him to refer to a particular one of those jobs. The operator can refer only to a running job, either by User_id or absentee slot number, when using the abs bump command. If a user has more than one job running, the slot number provides a way of referring to one of them, but there is no sure way for the operator to determine which slot "the" job is in - especially if they both have the same absin segment. And of course there is no way at all for the operator to refer to a job that has not yet started.

Attributes of a Useful Job Id

A job id should be a short, typeable, and pronounceable character string. Having the operator and user be able to use the same name for a given job is very desirable, so they can refer to it in verbal communication with each other. To avoid confusion and mistakes on the part of operators and users, it is almost essential that a given job keep the same job id throughout its life in the system, even across shutdowns.

The ids used by the operator (and hopefully by users too) must be unique among jobs currently in the system. However, ids of jobs no longer in the system should be available for re-use after some reasonable time, like a few days. This will prevent, for example, job numbers from getting progressively higher, longer, and harder to type, during the course of time.

Several alternatives were considered:

1. Use the unique id of the message segment entry as the job id. This fulfils many of the requirements, but it was rejected because of two serious flaws: 1) an 18-digit octal number is too long and hard to type; and 2) the unique id of a job must change if the job is reordered within a message segment or is moved to a different message segment.

2. Use the unique id of the message segment entry, and provide mapping of it into separate ids for the use of the user and the operator (by assigning ids at startup time for the initializer process, and once per process for users, and having them displayed by an absentee listing command and remembered in static variables.) This has the drawbacks that ids change at process termination and system shutdown, and users could not communicate requests about their jobs verbally to the operator. This might not seem serious, but imagine a user who knows a job by one id, who is unable to log in to execute commands to find out its current id or cancel the job, and is unable to ask the operator to cancel it because the operator knows it only by some other id. Another drawback of this method is that it forces us to design, implement, and debug two unique id assignment algorithms having different criteria for determining uniqueness.

3. Assign each job an id consisting of a short (4 or 5 digit) number. These ids would be reusable and would be assigned by an algorithm that chooses randomly from among the available ids (avoiding, by the randomness, the write down path that would result from assigning ids in numerical order).

   The id must be assigned by the initializer process, on receipt of the wakeup from the enter_abs_request (ear) command, or by an innner ring procedure called by ear in the user process. The id must either be published in a segment readable by all processes or be put into the queue entry of the job, requiring that the entry be rewritten by the initializer process, if that process assigns job ids. (New or modified rewrite primitives will be required to do this.)

   Assigning the id in ring 1 in the user process would require too much knowledge of the application (the absentee facility) to be put into what is now a general purpose system function (the message segment facility). So the initializer process will assign job ids. Publishing the ids in a segment readable by all processes would introduce an obvious write down path. Therefore the ids must be written into the queue entries.

The third alternative is obviously superior, and has been chosen.

7. Users will want to find out the current states of their jobs. Actions taken by the operator or the system can change the states of jobs - for example a job can be held for various reasons, and the order in which jobs are to be run can be changed. If users are to be able to see these things, state changes made by the system and the operator must be noted in

places accessible to the users - either the queue entries for the jobs (each of which is readable by the submitter), or in a segment readable by all users. Unfortunately the latter surely introduces a write down path, while in the former, serious implementation difficulties, inherent in the current design of the message segment facility, make it difficult to support reordering of jobs and still provide the ability for users to see the current positions of their jobs in the queue.

To properly support reordering of jobs and make it visible to users, we would need new message segment primitives that allow reordering of messages (more precisely, interchanging two messages, given their unique ids, by rethreading them while the message segment is locked), and a change that allows users with s and o access to be told the positions of their own messages. The problem is that the searching and salvaging procedures assume messages are in increasing chronological order (the unique ids are based on the clock and are increasing), and providing the ability to reorder messages would require either changing this assumption - probably prohibitively expensive to implement, and so not to be considered any further, or exchanging the message ids after rethreading the messages, which introduces another set of difficulties. If the message segment facility allows the changing of the correspondence between a unique message id and the contents of a message, then all system and user applications must be prepared to have such changes occur, and they can not depend on a given unique id always belonging to the message that it originally belonged to. While this might not be a real problem, especially if reordering is confined to the absentee queues, and possibly to privileged processes, prudence dictates a careful study of the implications of providing message-reordering primitives. Therefore they will be unavailable for use in early versions of the absentee enhancements.

Since reordering of queue entries is currently impossible, a general job-reordering capability could only be provided by having the initializer process maintain a list of jobs in the order they are to be run. Operator commands could modify this list directly. Since users could not be given access to modify this list, user-reordering could only be supported by means of requests sent by user processes to the initializer process.

Users can not even be given access to read this list without introducing a write down path. If users cannot see the current order of their jobs, nor the effects of the user's or the operator's reordering commands, the usefulness of user reordering is doubtful, and the wisdom of providing operator reordering is also questionable. Therefore we have decided to put off the implementation of a general job reordering capability until the possibility of reordering messages within the queues has been studied carefully.

To meet some of the job reordering needs, we will provide user and operator commands to move a job from one queue to another, and operator commands to force the immediate starting of a job and to move one or more jobs to "the head of the line." The implementation of the head of the line will be an additional message segment, queue zero, serviced ahead of the other queues, readable by all processes, but writeable only by the initializer process.

To make job status information available to users, we will rewrite the queue entry of the job whenever the status changes. For most jobs, this will occur only once, when the job starts.

8. To allow an RJE daemon to identify the jobs it submits as coming from a particular one of the several RJE stations that can dial to it, we propose the addition of a comments field to the absentee message in the queue. The comments field will be used by RJE to hold the name of the station. Since this field is settable by "the user" who is, in this case, the RJE daemon, the system will not trust its contents for purposes of making decisions about the job. However, the RJE daemon can trust this field and use it for purposes of controlling the jobs that it has submitted. Note that the RJE daemon can control no other jobs except those that it has submitted, and it has complete control of the comments fields of those jobs.

   Further, since the site (presumably) trusts the code running in the RJE daemon process, it can also trust the contents of the comments field of any proxy job submitted by the RJE daemon, and can instruct the central operators to trust it when issuing job control commands. (The thinking here is that, while the system should not contain code that automatically does such things as assign a job to a special load control group, bill it to a certain account, etc., on the basis of the contents of the comments field, it is safe for a site to instruct the operator to issue commands to do such things as hold or release jobs whose comments fields indicate that they come from a certain RJE station.

9. We have stated a need for an absentee job that requires some resource (such as a tape drive) to be so identified, and the job not logged in until the resource is available. On the surface this seems like a reasonable, and easy to satisfy, request. However, the more one thinks about it, the more difficult it becomes.

   If we wait until a slot becomes available and a job is ready to run, before we request a resource reservation on behalf of the submitter, then we have several alternatives. If the resource can be assigned immediately there is obviously no problem. If it cannot, then we could: hold the job and try for an immediate reservation each time a job logs out and we are about to log another one in; (2) or make a reservation for some time in the future and then: hold the slot unused for some arbitrarily long time until the resource becomes available; or else wait until the resource becomes available and then: hold the resource (charging it either to the user or to overhead) until an absentee slot becomes available (and hoping one becomes available before the reservation expires); or log the job in immediately, creating a new slot if necessary (the way the abs run command will do), thus allowing a job with an associated resource reservation to violate absentee load control and possibly overload the system; or invent the concept of a reservation for an absentee slot.

Once we do the latter, there is no reason not to let users use it: instead of having the system, at the time the job reaches the head of the queue, reserve a tape drive and an absentee slot for some time in the future and run the job then, the user could make that reservation at the time he submits the job, and have the advantage of knowing when the job is going to run. The big problem is how to reserve absentee slots without having to create new slots, violating load control as a matter of course, in order to meet reservation commitments.

A reservation of any resource is for a specified period of real time. Up to now, an absentee job has been a reservation of a process for an indefinite amount of real time, and a guarantee (barring crashes) of up to the specified limit of cpu time. An interactive login, on the other hand, is a reservation of a process for a real time period equal to the user's grace time, with no guarantee of any minimum amount of cpu time. In the interactive case, the user is present, and can see how slow the system is, and can decide whether or not to attempt to complete some large computation, or wait until later, or submit an absentee job to do it.

The only way we could reasonably impose real time limits on absentee jobs (thus allowing us to give reservations for absentee slots) would be to give logged in absentee jobs a scheduling priority sufficient to guarantee that they can get up to their cpu time limit during their real time limit. (The latter would be assigned automatically as a function of the former, and the scheduling priority would be some reasonable constant.)

(2) In the initial implementation, we will do this, since immediate reservations will be the only kind supported by the reserver. However this is an unsatisfactory solution since there is no guarantee that the job will ever run. This will become a real problem in later releases, when users can make reservations for the future, at sites whose resources are so heavily used that a reserver is really needed. The rest of this discussion pertains to that situation.

If we decide to go ahead with a reservation system for absentee slots, we will have to resolve conflicts between it and the other absentee scheduling algorithms (e.g., what to do if a job which the reserver has scheduled to run at a certain time has parameters (such as queue or cpu time limit) that conflict with limits that the operator has set).

To summarize, in order to associate a resource reservation for a future time with an absentee job, we really need to reserve an absentee slot for the time of the resource reservation. To reserve absentee slots we need to impose real time limits on absentee jobs, and to do that we must give absentee jobs a high enough scheduling priority to allow them to get their cpu time limit during their real time limit.

10. A method of guaranteeing the availability of absentee slots for such applications as transaction processing must be designed. There are a number of points to be considered in providing this capability.

The real need, for transaction processing, is for a facility that lets one process spawn additional processes to perform tasks designated by the spawning process. Since absentee is such a facility, we naturally think of extending it to fill the needs of transaction processing. But perhaps that would be misusing absentee for something for which it is not suited, and it would be better to provide a new mechanism for process spawning.

On the other hand, the absentee facility is already there, and if some natural and compatible extensions to it could be found that would allow it to fill the immediate needs, it would clearly be better, for reasons of economy, to avoid building a brand new facility.

Consider interactive logins and absentee requests from the following viewpoint: an interactive login is a request from a user for the immediate creation of a process that will take its input from a terminal, have a certain scheduling priority, and be charged at a certain rate. This request will be satisfied if load control permits it. An absentee request is a request from a user for the creation of a process at some future time; the process will take its input from a segment, have a scheduling priority usually lower than interactive, and be charged at a rate usually lower than interactive. This request will be satisfied by the system at its own convenience, when it has available capacity. The two attributes in which absentee jobs differ from interactive processes are: 1) an absentee job is the automatic creation of a process, by the system, to perfom a predefined task without the user's supervision; and 2) since the job is to be run at the convenience of the system, to fill up otherwise idle time, it is charged at lower rates.

There is no reason, other than historical, why these two attributes have to be tied together. Users might want the convenience of the first, but want, and be willing to pay for, higher priority than is implied by the second. In fact, that is exactly what is needed to support transaction processing.

A simple way of providing this within the current absentee facility would be to allow a site to designate one or more of the lower numbered queues to be governed not by the (current and new) absentee load control, but by interactive load control. Jobs from the queue(s) so designated would be logged in if there are primary units available in that user's load control group, without regard for whether or not there are any absentee slots available. The job would be charged to that load control group as if it were an interactive job. Such a job would never become secondary (subject to preemption).

To avoid any ambiguity regarding how a given job should be trated, and to preserve all of the features of the current absentee facility, it seems to be better to create a new queue for high priority jobs instead of using one of the existing ones. We will do so, calling it the foreground queue. The jobs in it will be called foreground jobs, as opposed to those in the other queues, which will be called background jobs. The term foreground process will be used to refer to a process that can be either a foreground absentee job or an interactive process. Jobs from the foreground queue will be charged at the same rates as are interactive processes. The load control for background absentee jobs will be modified to be consistent with, and compatible with, the load control for foreground absentee jobs. The details of these changes are discussed below.

This will cause some changes in which jobs can log in. Some jobs that currently can log in under certain system loads will be unable to. These will be the jobs of users whose load control groups are full. But in the current system, the fact that users whose groups are full can log in absentee jobs allows them to violate the spirit of load control. Denying them this ability should be viewed as a good thing.

Some jobs that could not log in under certain system loads will be able to, under the new system. These will be jobs from users who prefer to have the system perform a predefined set of computations for them without their supervision (i.e., absentee rather than interactive), and want it done immediately (or at a specified time) and are willing to pay higher rates for this priority, and whose load control groups have enough available units to enable them to log in interactively to perform these computations if they had wanted to. It seems obvious that providing users with this ability would be a good thing.

However there are some problems. We have not completely defined the behavior of the absentee facility with respect to jobs from various groups, in various queues, under various loads, and we must do so in a way that makes the system behave sensibly in all cases. For example, we don't want a job from the foreground queue to be refused login because the user's load control group is full, but the same job, from the same user, from one of the background queues, to be logged in because there are free absentee slots.

Of the two aspects of absentee jobs - predefined tasks run without user supervision, and low priority tasks to be run at reduced rates at the system's convenience to fill idle time - we have separated out the first, allowing such tasks to be run as foreground jobs. We now want to preserve, and possibly extend, the second.

Currently the existence of idle capacity is defined by the per shift parameters abs maxu and abs maxq. Thus, a certain amount of idle capacity, to be used up by background absentee jobs, is defined to exist on each shift, in spite of the fact that the actual load may cause a higher or lower idle capacity to actually exist. In a sense, the idle capacity is also defined by the load because, under heavy load, jobs take more real time to complete, and thus fewer jobs are run during a given time period. However, leaving background absentee jobs in the system for long times, competing for, and occasionally getting, cpu and memory resources that high priority processes also want, is probably not the right way to distribute the resources. It would be better, when high priority users load down the system, to finish any background absentee jobs in a short time, get them out of the system, and not log any more in until the high priority (both interactive and foreground absentee) load goes down. In other words, the number of background absentee users should be some function of current load level, varying between a site settable maximum and minimum.

The new absentee load control algorithm should obey the following rules: A user who could log in as a primary interactive user should also be able to log in a foreground absentee job. A user who can not log in as either a primary or secondary user should not be allowed to log in any kind of absentee job. But what about users who could log in as secondary interactive users? The system permits them to log in interactively, to fill the idle foreground slots that are reserved for primary users from other groups. However they are subject to preemption whenever the slot is needed by a primary user. It seems clear that most absentee jobs - either foreground or background - should never be made subject to preemption, since there is no user present to exercise judgement about whether or not to attempt a large computation in a secondary process, or how to interrupt such a computation gracefully if the process is preempted. Most users would prefer that their absentee jobs not be logged in until they can be run to completion. For those jobs that do not have a problem handling preemption, a -standby argument to the ear command, indicating willingness to be logged in as a secondary foreground user, could be provided. A preempt signal, analogous to the trm_ and sus_ signals now sent by the answering service (3) could be implemented, to allow preempted foreground absentee jobs to attempt to interrupt their computations gracefully, clean up, and log out.

Foreground jobs unwilling to be secondary could find themselves held while the same job in one of the background queues would be logged in and not be subject to preemption. Users would (correctly) see this as a bug in our load control algorithm, requiring them to play the frustrating game (familiar to supermarket shoppers) of trying to guess which line will move faster. To avoid this situation, we will allow such a job to be logged in if there is a background absentee slot available for it to occupy. The algorithm for deciding if there is an available slot will be described below, together with some new background absentee load control features that it interacts with.

With users potentially able to have several foreground (either interactive or absentee) and background processes simultaneously, more precise controls on the number of processes per user are needed. We currently have the multi_login (multip) attribute, which, when given to a user, removes the limit of one interactive process, leaving the user limited only by the number of process slots available to the load control group. Absentee users are always limited only by the number of absentee slots. To provide for proper administration of the new features, we must replace the multip attribute with two numbers, giving the maximum foreground and background processes respsctively. These will be per-user attributes, kept in PDT entries. Their values will be subject to per-project limits kept in SAT entries. In addition, we will provide per load control group parameters giving the largest fraction of background absentee slots able to be occupied collectively by users in each load control group. Thus users attempting to log in multiple processes of any kind will be governed both by the per-user and per-project multi-process limits, and by the load control group limits.

NEW FEATURES AND SCHEDULING POLICIES

(3) The answering service that sends these signals (to indicate process termination and channel disconnection) is not yet installed.

Currently, we have per shift abs_maxu (max absentee jobs) and abs_maxq (highest numbered queue being served) parameters, settable by the site, and operator commands to override the site settings temporarily. We will make abs_maxu a function of the foreground load rather than a constant. To reserve absentee slots for the lower numbered queues, we will reserve a (site-settable, per-shift) fraction of abs_maxu for each queue, and always allow jobs from lower numbered queues to occupy slots reserved for higher numbered queues. For example, if the slots per queue resulting from these computations were 1, 1, 2, then four queue 1 jobs could be running, or three queue 2 jobs (leaving one slot open for a queue 1 job that gets submitted while they are running), or two queue 3 jobs (leaving two slots open for queues 1 and 2).

The abs_maxu figure, and the per-queue reserved slots, will all be allowed to vary between a minimum and maximum value, as a function of foreground load. A formula of the following form will be used:

```
n = min(max_n,max(min_n,PCT*N))
```

where N is the input value (e.g. the number of idle units), n is the output value (e.g., the number of background absentee jobs that should be allowed to be logged in simultaneously), and PCT, max_n, and min_n are per-shift parameters settable by the site.

The complete set of formulae involved in these computations is:

```
system_maxu = f(config array in installation_parms)
available_units = system_maxu - n_daemons
idle_units = available_units - sum(units used by all groups)
recent_idle_units = (lowest idle units during last M minutes)
abs_maxu = min(max_a,max(min_a, PCT_a*recent_idle_units))
guaranteed_units(q) = min(max_gu(q),max(min_gu(q),
     PCT_gu(q)*abs_maxu))
```

where M minutes, PCT_a, max_a, min_a, PCT_gu(q), min_gu(q), and max_gu(q) are all settable by the site, all but the first per-shift, and the last three per-queue as well as per-shift. These new parameters will be kept in installation_parms.

What follows is an example constructed by plugging typical numbers into the above formulae.

```
       system_maxu = 85
       n_daemons = 7
so     available_units = 78

       sum (units used by all groups) = 22  (all interactive)
so     idle_units = 56
```

```
let     PCT_a = 10
so      PCT_a * idle_units = 5.6
        rounded, it becomes 6.
```

let max_a be 6.

so abs_maxu = 6, with 22 interactive users logged in.

(With only 13 users, PCT_a*idle_units = 6.5, which would round up to 7, but the absolute limit of 6 absentee users holds here.) Now, for this shift, let PCT_gu(*) = 20, 30, 40, 30; min_gu(*) = 1, 0, 0, 0; max_gu(*) = 2, 2, 0, 0; and min_a = 1;

The following table shows how abs maxu and the per queue reserved slots would vary with load, given the above parameters. The figures for abs maxu of 7 and 8 are shown, even though the max_a limit of 6 would prevent them from being used, in this example.

Users	Idle	PCT_a*Idle	abs_maxu	Q1	Q2	Q3	Q4
74	4	.4	1**	1**	0	0	0
73	5	.5	1	1	0	0	0
64	14	1.4	1	1	0	0	0
63	15	1.5	2	1	1	0	0
54	24	2.4	2	1	1	0	0
53	25	2.5	3	1	1	1	0
44	34	3.4	3	1	1	1	0
43	35	3.5	4	1	1	2	0
34	44	4.4	4	1	1	2	0
33	45	4.5	5	1	2	2	0
24	54	5.4	5	1	2	2	0
23	55	5.5	6	1	2	2	1
14	64	6.4	6	1	2	2	1
13	65	6.5	7*	1*	2*	3*	1*
4	74	7.4	7*	1*	2*	3*	1*
3	75	7.5	8*	2*	2*	3*	1*
0	78	7.8	8*	2*	2*	3*	1*

* These are the figures that are overridden by the absolute max absentees limit of 6 that was part of the example.

** These figures would have been zero except that setting min_a, min_qu(1) = 1 guarantees that at least one queue 1 job can log in, whatever the foreground load.

The per-queue figures in the above table are computed as follows: starting with queue 1, compute the figure using the formula for guaranteed units, round it, and assign the smaller of that figure and the remaining absentee slots; deduct the assigned figure from the remaining absentee slots; if there are more slots, proceed with the computation for the next queue. Notice that this gives preference to the lower numbered queues.

The algorithm for deciding if a job from a given queue can log in is as follows: reduce abs_maxu by the number of background jobs currently logged in; then, for each lower numbered queue, if the number of jobs from that queue currently logged in is less than the number of slots reserved for that queue, reduce abs_maxu by the difference; if the result is greater than zero then the job can log in. Notice that this means, for example, if three queue 1 jobs are logged in, we still reserve some slots for queue 2, at the expense of queues 3 and 4. In other words, when queue 1 wants to log in more jobs than it has slots reserved, it steals slots starting with the highest numbered queues, and works its way back through the lower numbered ones as necessary.

Earlier, we mentioned that a job from the foreground queue could log in as primary, even if the group has no primary slots available to it, provided that there is a secondary absentee slot for it to occupy. This can be determined from the above algorithm, if the foreground queue is assigned a place among the background absentee queues. The most obvious place to put it is ahead of queue 1. However, this might be unsatisfactory to some sites, where the queue 1 charging rates and scheduling priority are set higher than those of interactive users. Therefore, we will allow the site to set the place of the foreground queue among the background queues, by specifying, in installation_parms, the number of the queue it comes after. The default will be zero.

The abs run command (which allows the operator to start an absentee job immediately) will override absentee load control temporarily, and run the job even if a new slot has to be created for it. None of the absentee load control parameters will be changed by this command, however, and so the user will be charged with having a slot in use, and when the next absentee logout occurrs, all the slots will be seen to be in use, and a new job will not be logged in.

The abs suspend command will allow the operator to suspend a running absentee job temporarily, to decrease the load on the system. The mechanism that suspends an interactive process whose terminal channel has hung up will be used to suspend absentee jobs when the operator requests it. The most likely occasion for doing this is when all absentee slots are full, the abs run command has been (or will be) used to start an urgent job immediately, and it is desired to avoid overloading the system.

The abs move command will allow the operator to move a job to a different queue, in order to change its priority. Moved jobs will be placed at the end of their new queue. Moved jobs will be charged the prices in effect for their new queue. Moving a job into queue zero puts it ahead of all other jobs (except those already in queue zero). Such jobs will be charged queue 1 rates (the reasoning being that queue zero is just a device to get around the difficulty of reordering messages within a queue, and if this were not a problem, those jobs would have been moved to the head of queue 1).

To prevent large jobs from tying up background absentee slots during busy times of the day, we will add a per shift max cpu time limit for absentee jobs. Jobs that specify a limit higher than the maximum for the shift will be held until a shift with a higher limit arrives or the operator raises the limit. The abs run command will allow the operator to override this limit for individual jobs (i.e., a job named in an abs run command will be started even if its cpu time limit exceeds the maximum for the shift).

A job that was identified by its submitter as requiring a certain resource (such as a tape drive) will be held until the resource is available. The discussion under DESIGN ISSUES says more about this. Assuming the problems mentioned there can be solved, and a suitable interface between the reserver and the absentee facility can be designed, the abs run command will be able to override even a hold placed on a job because of a need for a resource reservation. In this case the operator will be queried, but a positive response will cause the job to be started even though it claims it needs a resource.

An abs hold command will allow the operator to hold some (or all) absentee jobs. This, together with the other new operator commands, will allow the operator to schedule all absentee jobs manually if he wishes.

With all the automatic and manual controls affecting when jobs are run, it is impossible to give any guarantees to users about the order in which their jobs will be run. Yet we are considering providing a facility whereby users can request that their jobs be run in a different order from that in which they were originally submitted. There is some question about the need for this facility. However, if it is provided, it will not be advertised to users as guaranteeing an order of running. It will merely allow jobs to switch places in the queues. The reordered jobs will still be subject to all the new scheduling algorithms. Users who require computations to be performed in a specific order should make them part of the same absentee job or set up a series of jobs where each one submits the next one just before logging out.

NEW AND MODIFIED COMMANDS

Modified User Commands: New Arguments

```
enter_abs_request  (ear)  -reserve  (-resv)  RESERVATION  STRING
     -foreground -notify -id
```

where RESERVATION STRING is a string acceptable to the reserver, describing resources needed by the job, -foreground is an optional alternative to the -queue N argument, causing the job to be placed in the foreground queue, -notify requests the answering service to send the user a message when the job starts and finishes, and -id requests the answering service to send the user a message indicating the id that was assigned to the job.

list_abs_requests (lar) NEW ARGS

where NEW ARGS are the same as for the operator command abs list (described below). They allow the user to list a subset of his jobs (or of all jobs, if -admin is used), and to control the order of listing and the amount of information printed for each job.

cancel_abs_request (car) JN {-force}

where JN is the number of a job to be cancelled. If the job is running, the user will be asked if it should still be cancelled. The -force argument will prevent this question.

New User Command

move_abs_requests (mar) JN1 ...JNn {-from_queue M} -to_queue N

where JNi are the numbers of jobs previously submitted by the user. They will be moved to the queue given after the -to_queue (-tq) argument. The queue given after the -from_queue (-fq) argument will be searched. If -fq is not given, the default queue will be searched. Only one queue will be searched.

Modified Operator abs Command Arguments

abs start

If the M and Q arguments are omitted, the default will be changed to be the abs_maxu and abs_maxq parameters set by the site, rather than being the constants 1 and 3, built into the code.

abs maxu {auto}

The word "auto" may be given in place of a numeric value to cause resumption of the computation of abs_maxu as a function of current load, using parameters specified by the system administrator in installation_parms.

abs bump JN
abs cancel JN

where JN is the number of a job to be bumped or cancelled. The bump command applies only to running jobs, and does not remove the job from the queue, while the cancel command applies to both running and queued jobs, and does remove the job from the queue.

New Operator abs Command Arguments

abs cpu_limit M

causes jobs whose cpu time limit is greater than M minutes to be held until a shift change or a new abs cpu_limit command raises the limit. The word "auto" may be given in place of a numeric value to cause the cpu limit to be reset to the value specified by the system administrator in installation_parms.

abs qres {r1 {r2 {r3 {r4}}}} {auto}

where rI are the number of slots to be reserved for queues 1 through 4 respectively. Omitted values are set to zero, causing no slots to be reserved for those queues. The word "auto" may be given as the only argument, to cause resumption of the computation of the per-queue reserved slots as a function of the current load, using parameters specified by the system administrator in installation_parms.

abs run JN

causes job number JN to be logged in and run immediately, in spite of any conditions that would normally postpone the running of the job.

abs suspend JN

causes job number JN, which is presently running, to be suspended. The abs release command takes the job out of the suspended state.

abs hold JN1 ... JNn OR -all OR Person.Project OR -comment STRING

causes the specified job or jobs to be held (i.e., not logged in) until subsequently released. Either of Person or Project can be *.

abs release JN1 ... JNn OR -all OR Person.Project OR -comment STRING

causes suspended or held jobs to be released, allowing them to proceed normally.

abs move JN1 ... JNn {-from_queue M} -to_queue N causes jobs JN1 through JNn to be moved from their current queue to the tail of the specified new queue. The new queue can be zero, causing the jobs to be run ahead of queue 1 jobs. If -from_queue is omitted, the default queue is searched. Only one queue is searched.

abs list {selection args} {sort args} {-brief} {-long} {-pathname}

lists the selected jobs. By default, all jobs are listed. The -brief argument causes the minimum amount of information to be printed: job number, queue position, User_id, absin entryname. The -long argument causes all information to be printed. By default, the information printed for -brief, plus some flags indicating the status of the job (e.g., deferred, held (and for what reason), running), and the comment field, if it is nonblank, are printed. The -pathname argument can be used with the -brief or normal modes to cause the full pathname of the absin segment (as opposed to just the entryname) to be printed.

The selection args are used to select the jobs to be listed, and can be chosen from the following:

```
JN1 ... JNn
-first N
-last N
-user Person.* OR Person.Project OR *.Project
-comment STRING
-cpu_greater M
-cpu_less M
-running
-eligible
-deferred
-held
-reserve STRING
-queue N
```

The sort args are used to control the order of listing and consist of the control argument -sort, followed by a keyword selected from this list:

```
comment
queue
cpu_time
person
project
```

The default is to list jobs in the order in which they appear in the queue - that is, the order in which they are to be run, except that held jobs (not deferred jobs) are bypassed instead of being run in the order shown.