Using a Task Manager to Improve User Productivity

Barry Polhemus
ETC Corp.
284 Raritan Center Pkwy.
Edison, NJ 08818-7808

Introduction.

The HP3000 is reasonably 'programmer friendly' when it comes to application development. Particularly if using TRANSACT/FASRTAN, it is easy in a prototyping shop such as ours for many applications to be developed rapidly as company needs arise. Most of our applications are transaction based and the user environment is highly interactive. The problem is, however, that as the company grows, the number of transactions also grows. As the needs grow, so do the number of applications. Soon, not having CPU horsepower to burn, the HP3000 could not keep up with our system needs. It became necessary to make some adjustments in the way work was being done on the system. There were many steps in our ongoing solution including spreading applications across multiple Series 70 machines. What I will present here is one of the steps taken to make our system (and users) more productive by decreasing interactive user time.

Many of the functions performed in applications can be separated into foreground and background tasks. By background task, I don't mean a batch process which could be handled by a periodic job stream, but rather a function which must be performed in a short time frame. These functions or tasks obviously require user initiation but do not require a user to sit in front of a terminal until the task completes. If the user can put together necessary information to perform a given task in the form of a work request, and simply submit that request to a background process to have the work carried out, the user becomes more productive. The time spent waiting for a CPU intensive task to complete can now be spent doing other work.

Drawbacks of a Simple Solution.

The design of a system to function in this manner could be built around MPE message files, but there are distinct disadvantages to relying exclusively on message files. First, let's look at this simple approach. There will be a submit process which puts together a work request and writes it to a message file. There will also be a background process to read the message file and carry out the requested task. The work request will be lost if the background process reads the request from the message file but is aborted before completing its task. If you have a means of determining what request was lost, you can requeue it, but there may be problems if the application requires the order of requests in the queue to be maintained. To avoid losing requests, we could first do a non-destructive read followed by a destructive read after the request has been processed. This technique will work

for a single background process, but not if you want multiple background processes to distribute the load for the same type of work request. There is no way to coordinate which process is working on which request.

Another diffculty will arise when you want to see what requests are pending for the background process. You can't just read the message file (eg. FCOPY) since that will wipe out your work requests. We can use COPY access to look at the message file which does non-destructive reads, but this requires exclusive access to the file. This means the background task which reads the file can't be running at the same time. Then there's the problem of starting and stopping the background process. If it's running in job mode, how do you let it know that you want it to stop without simply aborting it? If you write a request indicating self termination, how do you get it past the other pending requests in the message file? Reordering the requests in the message file would be difficult at best while the process is running. Actually, in addition to a termination request, the ability to reorder requests might be might come in handy if we needed to move a priority request to the front of the queue.

For a single application there could be a single background job to process requests, but when our number of applications increases to ten or twelve, we would get twelve separate jobs being fed by twelve separate message files each with their own set of maintenance difficulties as described above. As you can see, the number of messy details in our simple solution is increasing rapidly. Let's consider an alternate, more practical approach.

The Real Solution.

Even though message files have their drawbacks, they can be useful if used with discretion. Let's say there will be a submit process which will format a work request and write it to a message file. There will also be a background process to perform the intended task, but now it will be a son process running under a father/monitor process in a single job. The monitor process will act as a work request buffer between the submitting process and the process which does the work. It will pick up the incoming requests from the message file and store them internally. Even though there can be many submitting processes, there will be only one receiving process. The delay between writing a request and having it read is now minimal. We don't need to rely on the message file to hold requests until they are processed and we don't have to do any tricks to get the request from the message file while worrying about whether or not the work request will complete successfully.

The monitor will automatically maintain the order of the incoming requests as well as keep track, since there can be multiple son processes, of which requests belong to which son processes. Son processes can each perform a different function and/or they can

be duplicate sons used to distribute processing of a given function. There will be a separate queue for each son process and now that we no longer have the message file restrictions, managing the queues is now a much simpler task. We can now list the individual queues, reorder requests within a given queue, and move requests between queues. Of course, the latter may not make sense unless the source and destination queues handle the same type of request.

Queue management also involves starting and stopping individual queues. We need to selectively change a given queue's status without bringing down the whole monitor system. To accomplish these queue management functions, we need to send commands to the monitor from the outside world. Outside here refers to the inter-session environment. MAIL intrinsics suffice for father/son (intra-session) communication, but we need to send commands from an interactive session to a background process running in job mode. So we must go back to our trusty message file only now we don't have to wait for all preceding requests to be processed before our command gets through.

Since input to the monitor process may be an incoming request or some sort of command, we need some minimal structure to the incoming message. Each must contain a command identifier followed by data such as process number, request number, or whatever might be pertinent to that command. Incoming requests will be regarded as commands meaning pick up the request buffer and store it. As for requests, however, a little more flexibility incorporated into request routing will go a long way later. The need is simply to associate an incoming request with a particular son process. Good style suggests that we avoid things like hard-coding destination son process names into our submit programs since if we change our son process name, we also have to change our submit program. Also, in cases where there are multiple son programs to distribute load, we need a mechanism to submit requests such that our submit process doesn't care how many son processes might be available to perform the requested work. Even more important, we don't want to require our users to follow the load and explicitly submit to a given son process in the group.

The 'Chain' Mechanism.

We can achieve the flexibility we desire by using a method which I will call chaining. The mechanism will work as follows. Submit processes will put an identifier called a chain id in the request. There will be an external reference (chain file) which will relate a given chain id (request) to a specific son process number. The monitor will deal with its son processes by number. There is another external reference (configuration file) used by the monitor which contains program file names and provides an association between program file and process number. This way, the chain file as well as submit program are independent of physical file names. As the monitor receives a request, it will

look up its chain id and queue it to the appropriate son process. The chain file may indicate a single son process number or a list. In the latter case, the monitor can chose among the list of potential recipient sons based on which are active and how many requests are pending for each that is active. Thus load balancing among active duplicate son processes is automatic. Since we can start/stop specific son processes, we can regulate the throughput for a given request function. For example, if we have three son processes to handle a given request type and all are active, we can have requests distributed automatically between all three. At times when the load is lower, we can shut down (stop) one of the three so that requests will be distributed only among the two active sons. Also, since we can move requests between sons, any pending requests for the process we shut down can be moved to one of the remaining active processes.

Perhaps the most significant feature of the chain mechanism is also the reason for the name 'chain'. In the chain file, along with each chain id there can be a second chain id called the 'next chain id'. When a request is completed, the next chain id, if there is one, will become a chain id on the same request as it goes back to the monitor. This means that requests can be 'chained' from one son process to another automatically using the same physical requests. Thus it is possible for a son process to insert additional data into the request before it gets passed on to the next son. We now have the freedom to modularize background processing in any way we chose. Requests may follow different paths through the same group of son processes. For example, say we have three son processes through which a given request will follow through in sequence. We also have a similar request type which needs only the first and third sons. By simply supplying the appropriate chain id on the original request, we can direct it through either path via the chain file. Suppose we want to insert a new module in a given chain. All we need do is modify the chain sequence in the chain file and set up the new process in the monitor. We don't need to change any of the submit or associated son processes.

Communicating with the Outside World.

So far, I have only discussed the communication of information from the outside world to the monitor system (monitor process and associated sons). We also need a means of getting messages back to the outside world as to what's going on inside. There are two types of information we want to see. First, general information regarding the status of the system such as which son processes are active, how many requests are pending for each, the processing order and content of the pending requests, which requests are currently being processed, etc. Since all of this information can be kept in files dynamically updated by the monitor, all we need is a utility program which understands the structure of these files and can then display status information about the monitor system. We don't want any synchronization problems, so we won't allow the utility program to make changes

to these files. Should we want to effect some change to the system, such as shut down a particular son process, the utility program will send a command to the father process and the father will actually make the change. The utility program can then be used to verify that the did actually took place. The change won't necessarily be instantaneous since we're dealing with communication between two separate processes (actually, three in the example of shutting down a son).

The second type of information the monitor must provide are messages from the various son proceses regarding errors encountered while performing intended functions. That is, if the request didn't complete successfully, we want to know that it failed and probably why it failed. If son processes simply write error messages to $STDLIST the entire monitor process would have to be shut to check for errors. A more practical method is to log error messages to a file. This file could then be reviewed without disturbing the monitor process. Since there is usually no need to keep the error messages indefinitely, a circular file is a suitable choice. The simplest way to handle error logging is to have the father process do the actual writing to the logging file. The son processes can send a command containing the error message to the monitor which will then log it to the error file including a time and date stamp and process id. There can be a separate program to read the error log file and display error messages without affecting the monitor system.

Implementation.

Now that I have laid the groundwork for the monitor system, I will give some details of our implementation and describe more of the features of such a system. Though we are basically a TRANSACT shop, the monitor program (named Monitor/3000) was written in SPL for practical reasons. The son processes, as well as the submit processes can be written in any language. I will discuss these later. The monitor functions to link the submit process to the son process(es) which eventually carry out the given task by passing the work request from one to the other. Only the submit and son processes need understand the content of the request. The actual request does include some header information, however, on the request as received by the monitor such as a request indicator to identify it as a request as well as the chain id. The rest of the request is blindly passed along to the appropriate son.

There are several files used by the father process for storing requests, maintaining queuing information, maintaining status information, as well as the chain and configuration files mentioned earlier. The configuration file contains not only the list of program files that represent the various son processes but also contains the file names of the other internal files used by the monitor system. This way, all of this configuration information can be obtained by the monitor program via a generic file equation without hard-coding any file names into the monitor

program. We can now set up multiple monitor systems on a given HP3000 by simply setting different file equations. The associated utility program previously mentioned works in the same manner. There is only one physical utility program which gains access to the various monitor systems via a different file equation before running the program.

Mission Control - the Utility Program.

The control of the monitor system really lies in the utility program. It, too, is coded in SPL for practical purposes. By practical, I am referring mainly relative to TRANSACT, given what the program needs to do. The utility program gains access to all of the monitor system files via the configuration file back-referenced through a file equation. The utility program can display the overall state of a given monitor system including number of son processes, the current status of each, and the number of pending requests for each. The requests' content can be explicitly listed in octal and/or ascii formats. A given request can be prioritized by moving it to the front of the queue for its destination son process. As mentioned before, requests can be moved from one son to another (provided they can service the same type of request as per the configuration file). Requests can also simply be purged.

The utility program functions not only to manage requests but also the state of individual son processes. There are three main states in which a son process can be. First, there is Down, which means that son has no associated physical process (no PIN). There is Inactive, which means that son process has been created, but it is not activated. A son cannot process requests in either of these states. Finally, there is Wait which means the son has been activated and is waiting for a request to process. Once there is a pending request, the status will be Active, implying that a request is currently being processed. The contents of that request can be listed even while it is active. There is also a special case when that occurs when a son is started up with requests pending. This Busy status means that the monitor process has sent a request to the son but the son has not yet received it because it is busy going through its start up procedure.

Utility program commands exist to change a given son between the three states. There is CR (create) to go from down to inactive, and SU (start up) to go from inactive to active. These correspond to MPE Intrinsics CREATE and ACTIVATE. There is SD (shut down) which moves a son from wait/active to inactive. The currently processed request, if any, is allowed to complete. During this time while the request is completing, the status will appear as Shut Pending for that son. As it completes, the son is suspended. There is also KI (kill) which removes the son immediately from the system. If there is a request being processed it is left in the queue.

Before continuing, let me digress a bit and discuss queue integrity. The first concern is in not losing requests. The monitor system I am describing meets that requirement. Since it stores the requests internally, the requests remain intact until they are either completed as determined by the monitor itself via notification from the son which processed it, or an explicit purge request is received. Whenever a request is being processed and does not complete normally due to the son process being killed, the entire monitor process being shut down or even aborted, or even a system failure, that request is left in the queue and will be dealt back to the appropriate son when the system is restarted. (Of course, this is under 'semi-normal' circumstances, there isn't much that can be done for disc crashes, for example.) The other concern with queue integrity is maintenance of chronological order of requests. In most applications this is not a necessity but in others it can be vital that requests be processed in the order submitted even if not processed right away. Again with queue integrity in mind, our monitor design includes periodic (whenever a change takes place) posting of queuing infomation to disc. Thus the individual queues remain intact even if the entire monitor system is aborted.

There are a few remaining utility program commands which I will mention. BR (break) can be used to abort processing of the current request and have the son continue with the next request. GO and SH are the commands which start and stop the entire monitor system. There are also some commands not for use by the average user such as those to initialize (IN) all the queues (wiping out all requests for all sons) and a renew (RN) command which rebuilds all the queues from scratch but destroys chronological order. VR (verify) causes the monitor process to do an internal verification of its queuing buffers and lists. These commands are shielded from the typical user by a security code. Another protected command allows the priority of individual sons or the father itself to be changed among CS, DS or linear queues. There are some realistic checks built in here, however. You can't put a son in the linear queue at a priority higher than the father or at a priority higher than the the low end (highest priority) of the CS queue.

The remaining commands are harmless. There is a pair of commands which function like OUT=LP and OUT=TERM commands in QUERY. Output from the listing or status commands can be sent to a file or printer. The ID command will display version id and some configuration parameters. There are some semi-arbitrary limits on the monitor system such as the maximum number of sons that can be included in a given monitor system as well as the maximum number of requests that can be stored internally. In our system, we have set the maximum sons at 15 and the maximum requests at 2000. When a son process aborts (or is shut down), the pending requests will continue to accumulate until it is restarted. If enough requests back up, the monitor will no longer accept requests until room is made available by either purging requests or processing them. In the meantime, requests simply accumulate in the message file.

When the internal request limit has been reached, the monitor periodically displays a message on the console indicating a request overflow has occurred. Display of this message can be turned off or on via a utility command. Once you realize that an overflow has occurred, there is usually no need to continue the console messages. In some applications, 2000 requests may be much too many to take advantage of the overflow warning system. The number of requests that will be accepted by the monitor is configurable up to the maximum allowed, so if 2000 requests constitute a month of work, it is probably better to set the internal limit at more like 200 instead.

The Workers - Son Processes.

Enough said about the utility program. Now I'll discuss some aspects of developing/converting applications to run under the monitor, namely submit programs and son programs. The submit program has a simple task. It only has to collect any information necessary to perform the task in question, put it in a format that will also be understood by the destination son process, and write it to the request message file. The request has some header information (such as the chain id) followed by up to 480 bytes of data. In some of our applications, the request consists of only a file name but in others it may contain a fairly wide IMAGE record right in the request. Since both our submit programs and son programs are written in TRANSACT, it is a common practice to put the request buffer definition in an include file for use by both submit and son programs. This insures that no discrepancies exist in the request contents and tends to make programs more understandable because the same variable names are used.

The son programs need to carry out the intended task based only on information from the request. The physical process of request handling, ie. getting requests from the monitor (father) and letting the monitor know when the task has been completed, is provided in the form of a monitor interface. For our TRANSACT applications, there is a file which is to be included at the beginning of the program which does all of this dirty work. The interface code will do a perform to a predesignated label for each request it receives. It also performs to another label for initial item listing and one for any initialization code. The son program then only needs to include the interface file and provide the appropriate labels and need not be concerned with details of request passing. Once code is developed to perform a function, making it run as a son process is quite simple. This same style has been extended to other languages as well. We have one son application written in SPL and we have done some testing in PASCAL. In these cases, the interface is in the form of procedures placed in an SL or RL such that the main body of the program does nothing more than call the interface procedure. The son program will have one main procedure (other than 'main' body) for processing a request whose calling address is passed to the interface procedure so that it can be called from there. Three other procedures can be included, one for initialization upon

startup, one for processing upon restart after a shut down, and one for processing prior to a shut down. The latter of these procedures is only used in case of a normal shut down command as described above. For a normal shut down, the father process tells the son to suspend when it has finished processing its current request, if any, and the son can perform any shut down functions prior to suspending. If the son is killed, no shut down processing can be performed.

The design of a monitor process (submit/son pair) is basically separating the information needed to perform a task from the task itself. A typical application will include both an interactive section to specify what has to be done, and a section to do the work. In cases where the task is simple, creation of a monitor process may not be practical. However, a task which is CPU intensive relative to the interactive front end is a prime candidate. Another situation might be when the tasks are very simple but numerous and can be queued sequentially as a batch. Let me give an example to differentiate. Suppose there is a chain in an IMAGE data set and we need to perform some action for each entry in the chain. The submit program can queue just the key value in the request such that the son reads the chain and processes accordingly. The alternative is to have the submit program read the chain and send requests for each entry. The latter approach involves a lot more request traffic but for some applications this trade off is necessary. In either case, the user is supplying only the key.

One more comment about developing son processes concerns error handling. Different coding styles usually involve different degrees of error handling. Monitor son processes must be made as robust as possible so that if anything goes wrong during processing of a given request, short of major catastrophe such as a corrupt data base, the error will be handled gracefully and the son process will continue with subsequent requests. In the above example where the son process accepts a key value to read in an IMAGE data set, and the key value is invalid, the son process should just log an error message and move on to the next requestrather than abort. Error logging is as simple as putting pertinent information in a buffer and calling a procedure (or perform for TRANSACT) supplied as part the monitor interface. The more son processes abort on their own, the more attention must be paid to son process status and management. It is certainly more desirable for the sons stay running without paying attention to them, especially if there is a high volume of requests being processed. If a high volume son aborts, the incoming requests will back up quicjly and if not discovered soon enough, the backlog may be difficult to recover from. This is an example of why one might configure the total requests held internally by the monitor to less than the maximum so that if a son aborts and the queue backs up, the console warnings give an earlier indication that a problem exists.

Real-Life Examples.

I will now give an example of some of the ways we use our monitor system. We have a chemical analytical lab which generates data on a variety of intruments, predominantly HP1000 based mass spectrometers. Data files containing analysis results are transferred from the HP1000 to the HP3000 and subsequently a request is written to a monitor message file which contains only a file name. A son process receives this request and will load the file into a transitory review data base. Once the file is loaded sucessfully, the file is purged. The loading process looks up some pertinent information from another data base and stuffs it back into the request. The request is then chained to a second son which takes the added information from the request and performs some other cross referencing functions. This processing is all automatic.

The data is then reviewed in the temporary data base with interactive processes. There are batch oriented calculations done by a different son process which are also reviewed subsequently via an interactive process. When review is complete, data organized into groups which represent the final reports, and sent to another pair of data bases which will hold the data indefinitely. The problem is that these data bases are on another HP3000. The solution lies in a another son process that handles generic file transfer via a remote I ǂ session as well as simultaneous (part of the same request) writing of a supplied buffer to a message file on the remote system. The remote message file is, as you might guess, an input request file for a monitor running on the remote system. The request to the transfer son process contains a local file name, remote file name, remote message file name, and buffer to write to the remote message file along with some options such as overwriting the remote file and purging the local file after transfer. The program is designed to handle file transfer only, message transfer only, or both.

The user flags a series of groups in a given batch of results that are ready to be transferred and then queues the batch name to a son process which will find the flagged groups, put each group into a file, and queue the files to the transfer process process described above. That son process transfers the file to the remote system and writes a record to a remote message file which effectively queues a son process on the remote system to pick up the file and put the data away into the appropriate data base. These remote requests get chained to a second son process which print reports from the data in the data base. Since there is a significant volume of these reports queued directly from the remote system, we have two copies of the son process which produces the reports to avoid building a backlog. However, due to some improvements made to our report program, a single copy has been able to keep up with the load so that the dynamic load balancing trick mentioned earlier isn't really necessary. A third son process in the chain can take the data for the report and put it into our electronic mail system where our clients can

dial in and view their data before the hard copy reports are mailed.

Another of our applications for monitor processing involves data base synchronization bewteen two HP3000 systems. We have some master data sets (as in master/slave, not IMAGE master) on one system with mirror image, read only slave copies on a second system. Any time a change is made to master side, the changes are queued to the transfer process described above in the form of a file or for cases of a single record update, the whole IMAGE record is put in the request. The request that ultimately reaches the son process on the slave (remote) system is encoded as to type of update in terms of file, single record, add, update, delete, and what else is in the remainder of the request. The single record updates occur in very high volume but are processed rapidly including transfer as the transfer process not only remains logged on to the remote system as long as it's active but also leaves the remote message file open as long as the message file writes are to the same message file. We saw to it that this was the case to avoid a file opens (for the message file) on a per request basis, let alone remote logins. Since much of the synchronization involves client order changes, maintaining chronological order of the changes is critical as well as just not losing the changes. In other words, queue integrity is important even across systems.

Conclusion.

We have gotten a lot of mileage out of the monitor concept. We even have a monitor system on the HP1000 mass spec data systems mentioned above for processing raw data from the instruments, reporting the processed data, and archiving the raw data to tape automatically. As a matter of fact, our HP3000 monitor concept grew out of the HP1000 version. The physical mechanisms are as different as RTE is from MPE but the basic functionality is the same. The HP1000 monitor is an integral part of the Aquarius software package developed by ETC which is now comes standard with HP's RTE mass spec systems.

Even though the design I've described satisfies our needs, further extentions are certainly possible. A simple extention might involve capturing statistics such as the volume of requests and even processing time. It might then be possible to build in some 'smarts' to the monitor program to evaluate request load and automatically startup/shutdown son processes based on the load. Another extention might be to build in some time delay before the submitted request can be processed or, in effect, be able to time schedule certain requests.

We feel the monitor systems have been a tremendous success in making our computers do what they do best without making the computer users wait thereby increasing their productively. If the users had only computer tasks to perform we would probably be in trouble, but even with our great degree of computer automation, a

number of paper tasks still remain, and our users now have more time to devote to these tasks.