Process Handling for Fun and Profit

Jeff Kell
UTC Computing Services

Process Handling Fundamentals

A program with process handling capability may create, delete, and exert certain controls over other processes. Processes created by a program of this type are called son processes, the originator of the processes is called the father process. This relationship may extend to other 'generations' of processes as well.

In essence, a process handling program has the capability of doing a ':RUN' command. In the most simple form, a program creates some other program, suspends itself, and the newly created program begins execution. When the new program completes, the original program is reactivated. However, the father program does not have to suspend. It may continue executing some other task, or create additional son processes.

This concept can be applied to improve on many application systems, as will be explored later, but it also adds a degree of complexity to the operation. Most process handling applications can be duplicated, at least in terms of results, by standard programming techniques; but process handling can improve throughput, performance, and flexibility of certain applications. Thus, a tradeoff point is evident when the advantages of process handling cannot justify the extra code required to support reliable process handling applications.

Process Handling Difficulties

The most prominent difficulty in any process handling application is the need for error detection and handling, especially when more than one process is created, or the depth of process handling extends beyond the initial generation. Whenever the father process continues execution in parallel with the son processes, it is difficult enough to simply detect the completion of the son, let alone detect errors such as program abort conditions.

When only 1 son process is involved, and the father suspends while the son executes, the father will always be activated upon completion of the son process (or abnormal termination). Simple communication mechanisms such as a JCW may be used to determine the termination status of the son. When multiple son processes are involved, there is always the possibility of a son terminating while the father is already activated, in which case the activation has no affect; thus the father is unaware of the son's termination. There is no overall method of reliably detecting this condition. To overcome this situation, some form of periodic activation of the father can be performed so that the father can check on the status of each son.

Communication is often necessary between the members of a process handling family, if only to communicate status information. There are numerous ways of passing information, such as PARM, INFO, mail, message files, and data segments. The latter three allow for two way communication, and are usually the tools of choice. For example, a father process may have a message file for status reception. When a son terminates, the termination status and son identification are written to the message file. The father process can then read the message file until all sons have completed. If the father process uses 'timeout' reads on this file, which is possible, it can check for any abnormally terminated sons on each cycle.

When many processes are involved, the number of required message files may become too large

to be considered efficient. This becomes readily apparent when two way communication is required, as each son process must have its own message file. In such cases, a shared data segment is the most efficient communication mechanism. If this route is chosen, special procedures must also be written to support the transfer of messages through the data segment.

Theoretical Divisions of Process Handling

Process handling applications fall into one of two broad categories. The first category, independent processes, encompasses most typical uses of process handling. Independent processes do not depend upon other processes for their welfare. The second category, dependent processes, is much more complex. These processes work together to accomplish a common goal, and usually involve a high degree of communication and control.

Some applications do not fall into a specific division, and may exhibit characteristics of both. Independent processes have only a limited value in terms of processing advantages, but dependent processes are worthy of greater investigation. Applications which combine both areas are useful if the goal is to control several independent tasks, with the tasks being accomplished through dependent processes.

The most common form of independent process control is the menu driver. A menu driver program displays a set of command names and allows the user to select a command from the list by name, function key, or some other method. The menu driver, in turn, executes the program corresponding to that function. Menu drivers are often added to systems which are comprised of several separate programs to ease the transition from one program to another. The menu program in this case is suspended while the selected program executes.

Slight revisions to this facility could allow the user to select several "background" tasks to be executed while an on-line function is being done. For example, several reports may be requested while the user executes an on-line update program.

Menu programs often provide control information to the selected programs. For example, the menu program may set up file equations, obtain various run options, etc. In this way, extended control is provided over the selected programs.

The impact of such applications on the overall performance of the system varies on the implementation. In normal execution, MPE has a CI process for each user. When a :RUN is requested, the program is added as a son process of the CI. Thus, 2 processes are present for each user on the system. Thus, given a number of users (U), a number of user programs (P), and a fixed number of processes required by MPE (S), we can examine the number of processes required to support a given implementation of application systems. Note that other resources (data segments, etc.) also increase in proportion to the number of processes.

Without process handling, the number of processes is:

(a) U + P + S, or 2P + S

This equation holds for batch as well as on-line. Each additional user will require 2 processes. With a given physical limit of 256 processes under MPE, the limits on the number of users on a Series 64 (without MPE-V) are clearly defined by (256-S)/2.

Simple menu drivers (with a single son task) clearly complicate the matter, as both a CI process and a menu process will be required for each user. For this case, the number of processes is:

(b) 2U + P + S, or 3P + S

Menu drivers which allow multiple processes help somewhat by removing the CI process for each additional task, but the tasks must be batch oriented since the terminal user can only run one on-line task at a time. This equation is given by:

(c) 2U + P + S, or
(3P + S) - (batch tasks)

To really improve the system load, process handling must be extended to an additional level. Consider one main process which creates a menu program for each terminal, redirecting STDIN and STDLIST to the appropriate terminal. Now the equation is:

(d) U + P + S + 2
(2 is the main and its CI)

This equation is only 2 processes over the normal (a) equation, and allows a menu driver for each user. Since the main program controls access from each terminal, security and logon provisions can be added easily. If the attached menu programs are of the multiple type (for concurrent batch tasks), the reduction is even

greater. With no-wait I/O (priviliged), the main task could be a 'global' menu program, controlling all terminals. This eliminates the user menu altogether, giving:

(e) P + S + 2

With this implementation, well over 200 users could be supported on MPE-IV. In general, the process related resource requirements are cut by nearly half. A non-priviliged solution to the above model would be to provide communication paths between the main program and the son processes, then attach the user menu programs. When the function was selected, the request would be sent to the main program and the menu program can terminate. The main program would then attach the appropriate application program. Thus, the main program would alternate between a menu and application processes. The process equation remains the same, but more process switching is required (a suitable tradeoff to remain non-priviliged).

Dependent Processes

Process handling applications involving two or more processes working together for a common goal are the most beneficial forms of process handling. Exact implementations differ on a widespread scale, with only vague suggestions of a concrete definition. Dependent processes are best defined by examination of the basic concepts which can be used. Actual applications usually combine more than one concept, and may utilize only a basic theory rather than a literal definition.

The concept of dependent processes is difficult to define, as some areas are difficult to classify. For example, TDP uses process handling when formatting its output by creating the program SCRIBE to perform the formatting. From one standpoint, the two programs are working together for a common purpose; however, it would be possible to get the same results if SCRIBE was an independent program. TDP is using SCRIBE as a 'sub- routine' to format output, yet this formatting is independent of TDP itself. This arrangement is marginally classified as dependent by the first definition to follow.

Co-routines

A co-routine (usually written coroutine) relationship exists be- tween processes which, when combined, perform a complete task. In strict terms, coroutines are always in pairs, and may ar- bitrarily 'swap' execution from one to the other. A coroutine relationship simply implies a higher level of 'subroutine' than with subprograms. The application of coroutines leads to greater modularity and all of the advantages incurred by modular program- ming. Coroutines may be used to provide greater modularity, or to split a large task (possibly too large or inefficient for a single program) into manageable parts. Standard 'menu' applications described earlier could be considered in this category, but such applications rarely communicate with each other. Coroutines work together for a single transaction, while the applied programs of a menu are usually independent transactions or func- tions. The TDP/SCRIBE relationship can be loosely applied to either definition.

Pipe Processors

The first logical extension of the coroutine relationship is the pipe processor. If the processes of a coroutine are designed so that they accept 'input' from their predecessor, perform some ad- ditional processing, and send 'output' to their successor, the processes have been changed to pipe processors. Programs in this category have an input 'pipe' sending information to be processed, and an output 'pipe' transmitting results. Pipe processors can be hooked together in any number of ways by con- necting the output 'pipe' of one process to the input 'pipe' of another.

One application where this method can be elegantly applied is data base inquiry and report. In this case, there are usually a number of different retrieval methods, sorting methods, and print formats. If each unique retrieval method, sort criteria, and print format was coded as a pipe processor, a flexible reporting system can be created from a minimum number of programs. At run time, the desired retrieval method, sort sequence, and print format are selected and linked together. Each record selected by the retrieval procedure is passed to the sort routine by the pipe. The sort routine passes the records to the SORT procedure until all records are processed. The

sort routine then retrives records from SORT and passes them to the output process by the pipe.

In simple coroutine relationships, these processes would not overlap. Instead of passing data in the pipe, they would trans- fer control to the next process. When the next process completes its handling of the data, control is transferred back to the first process. In other words, the processes must be synchronized together with no overlap. Pipe processors must also be synchronized, but they may overlap. A pipe processor waits until data is available in the pipe, but begins processing once the data is read. When writing to a pipe, the process must wait until the next process reads the data before it loops back for more input. Some overlap is obtained, but the pipe can only hold a single record.

The sorting requirement places a bottleneck in this scheme. All records must be given before the sort process can release one to the report procedure. If the sort is not required, a greater overlap can occur. Without the sort, we have TRANSACTION overlap; with the sort, we have PROCEDURAL overlap. Some applications cannot overlap either case.

Transaction Pipe Processors

When transaction overlap is possible, pipe processors can perform at their most beneficial level. In all pipe processors and coroutines there is some synchronization to be performed. With coroutines, the lack of overlap implies the synchronization, much like a main program calling a subprogram. With pipe processors, there is the question of overlap: After writing results to the output pipe, when does the pipe processor go back for more input? In applications where there is transaction dependency, a transaction may have to finish the entire pipe before the next transaction can begin. In this case, we have no overlap at all. When transactions are independent, the pipe processors can proceed at will, limited only by the capacity of the pipe (in simple pipe processors, the pipe holds only 1 transaction). Overlap is pos- sible, but limited by certain restrictions.

In all cases where a transaction is split into pipe processors (or stages), the processing time for each stage will probably vary from one stage to another. To complicate the matter, the For purposes of present discussion, the average processing time will suffice, and thus the average time for the stages will probably vary.

For example, consider a given transaction divided into four stages. The average processing time for each stage can be denoted by the variables s1 through s4. Regardless of the actual time required for each stage, the time required for each transaction without process handling is:

```
Linear (no process handling):

  s1 + s2 + s3 + s4
```

The use of coroutines will result in a slightly larger transaction time, since the overhead of coroutine switching must be added to the time required for the four individual stages. The same rule applies to pipe processors with transaction dependence.

When transaction independence is possible, the average time for each stage becomes important. If each stage is equal, the time required for each transaction is the time required for a single stage, plus overhead of filling the pipe at first, plus emptying the pipe after the last transaction. The following illustration depicts this case, using 't' for transactions and 'T' for time:

```
Pipe Processor Stages          Time        Linear Processor Stages
  s1   s2   s3   s4             (T)           s1   s2   s3   s4
 ---- ---- ---- ----           ----          ---- ---- ---- ----
  t1                            1             t1
  t2   t1                       2                  t1
  t3   t2   t1                  3                       t1
  t4   t3   t2   t1             4                            t1
  t5   t4   t3   t2             5             t2
       t5   t4   t3             6                  t2
            t5   t4             7                       t2
                 t5             8                            t2
```

Of course, this example is an ideal situation. Several factors can interfere with this ideal case: stage times may be unequal, stages may interfere with each other, and stages may have idle time. If the stages are unequal, and the pipe only holds 1 transaction, some stage will obviously be idle for some period of time. This leaves only two main factors to examine: unequal stage times and stage interference.

If stage times are equal, the average transaction processing time is the length of a single stage (once the pipe is filled). If unequal, the transaction processing time is, at best, the length of the longest stage time. If more than 1 stage has an unusually long stage time, and the stages occur in a worst case sequence, the time may be slightly longer. There is, however, a definate overlap of the stages with shorter times.

Stage interference can be present if the stages are divided so that there is contention for some common resource between two or more stages. Examples of such cases include locking a common resource (data base, file, Rin, etc.) and heavily I/O bound stages. CPU bound stages may interfere if the need is excessive (the stage cannot complete before its allocated quantum expires). Most transactions are I/O bound, and CPU bound stages consequently fill in the gaps. If the CPU time for a transaction exceeds the I/O wait time, there is a probability of CPU resource contention; however, the interference caused by other resources is much more exaggerated. Exceptions can be found for most attempts to define rules for preventing resource contention, and to some ex- tent, there is always some interference. The only general guideline is to avoid gross interference cases such as locking.

Proper division of stages to eliminate interference leads to a pipe processor approach which can achieve overlap, which is clearly more efficient than the traditional linear approach. Once this goal is attained, optimization is simplified by con- centrating on the longest stage. However, this approach does not address the problem of data dependent processing time mentioned earlier.

Queued Pipeline Linkage

The next extension to transaction pipe processors involves the means by which the processors are linked, or the pipe itself. If the pipe is a queue instead of a 'mailbox' as assumed earlier, greater overlap is possible, especially when variable stage times are involved. Each stage proceeds at its maximum speed, limited only by the capacity of the queue. The queue effectively smooths out irregularities caused by variable processing times in most cases. The longest stage time still limits throughput, but this approach helps when the longest stage time is variable. Slower stages keep a queue of waiting data, while faster stages remain idle. Interference may be exaggerated by a queued pipeline, especially in I/O bound situations. Queueing will improve over- all throughput (in general) if the straight pipe linkage alone does not saturate the system. It is often necessary to impose some limit of transactions which are allowed in the pipe, espe- cially when pipeline applications affect interactive response time. Most queued pipeline applications can be tuned to a level where the CPU or I/O system is saturated; consequently, other ap- plications on the system will experience poor response time.

All of the previous techniques take advantage of the fact that some idle time is caused during transaction processing. Time which was previously idle is now spent on another task in paral- lel. Although the application under study is experiencing throughput increases, there is a greater load imposed on the sys- tem, and less time is free for other applications. To minimize this impact, the process handling application should take advant- age of resources not currently being utilized. On the other hand, if a given I/O bound application is slow because of the to- tal I/O load of all applications, process handling will only slow everything down. At this level of complexity (and subsequent levels) the tradeoffs in this area determine the benefit or cost of process handling applications.

Switching Pipeline Manager

A switching pipeline manager is a front-end processor for more than one pipe application. In general, if more than one depen- dent process handling application is in use, a switching pipeline manager can be used to route transactions to the proper application. Usually this type of manager controls transaction pipes, and the examination will be limited to this application.

Most systems involve more than one type of transaction, but it is desireable to stay in the same program. An independent process parallel is the menu program, defined in the first section. With a pipeline manager, however, the entire transaction is gathered and 'switched' to the input pipe of the appropriate pipeline. The user may begin the next transaction while the current trans- action is being processed. This is particularly useful in applications which require a printed document as a final result, such as inventory tags, order entry, etc. The user can begin the next transaction while the previous document is being printed. If more than one document is used, the pipeline manager controls the production of the

appropriate one (otherwise a simple pipe processor is sufficient).

The benefit of a switching pipeline manager is limited for online applications, but is extremely useful for batch applications where input (or requests) come from a common source. The next section deals with applying this concept to online systems.

Centralized Pipeline Manager

With online applications, it is obviously wasteful to have a copy of the entire pipeline system for each user. A centralized pipeline manager collects requests from all users of a given system and passes them to a common pipeline system. The collection procedure for the user simply validates the transaction and places it in the input pipe for the centralized manager. In ef- fect, all user collection programs write to a common input pipe (queueing is almost mandatory). If a single application is pipelined with a message file as input, the manager is not really required. A manager process can, if present, control the number of transactions in the pipe and queue the overflow. As mentioned earlier, this limit is frequently necessary.

If more than one application is being pipelined, the centralized manager can be merged with the switching concept described above. In this case, transactions are centralized from the different user terminals and then split according to transaction type. With this setup, it is possible for the manager to provide infor- mation about the number and type of transactions performed for each terminal. Security can be enhanced by limiting transactions by terminal. The manager can even log transactions for recovery purposes without the need for IMAGE or other logging code in the transaction processors. If certain transaction stages require a lot of information to be transferred from one stage to another, the manager can assign extra data segments (from a limited pool) to the transactions. Many functions can be done, but they imply some means of posting the completion of a transaction. This can be easily done by having the last processor in each pipe write a completion message in the manager's input pipe.

Vectored Pipeline Scheduler

In many pipeline applications it is desireable to select certain stages which apply to the transaction. Some pipelines may have common stages, in which case duplication should be avoided. For these cases, the stages can be identified by some method known to the scheduler and the collection program. As a transaction is collected (or received by the scheduler) it is assigned a vector of stages through which it must pass to complete the transaction. In this case, each stage posts completion to the scheduler, which in turn selects the next stage for the transaction. It is possible for each stage to be given selection logic so that sub- sequent stages can be invoked directly, but this causes some redundant code and disables the tracing and monitoring capabilities that can be included in the vector scheduler.

A number of additional functions can be included in the vectored scheduler, particularly when combined with the centralized con- cept previously discussed. The vector can be data dependent, and the vector can be modified by any of the stages. Stage modifica- of the vector can be used for error handling - if a stage detects an error, the vector is changed to pass the transaction to the error handling stage(s). The efficiency of a vector scheduler is much more important than other schedulers due to its intervention between each stage, rather than between transaction tions. For this reason, the vector scheduler usually maintains a high priority queue for stage pipes and a lower priority queue for incoming transactions.

Swapping Vectored Pipeline Scheduler

At the extremely complex end of transaction processing systems we find the swapping vectored pipeline scheduler, an extension of the vectored scheduler defined above. When the number of stages exceeds the number of processes which can be efficiently managed by the system, it is impractical to keep all possible stages loaded at once. The swapping version of the scheduler controls dynamic loading and unloading of stages as required or desired. From a simple requirement standpoint, a preset limit of stages can be determined as the maximum number to be loaded at a given time. As stages are needed, the scheduler fills the pipe (if loaded) or loads the stage. Some replacement method is needed to swap out stages to make room for additional stages when the limit has been reached. The scheduler can be modified to unload stages that are unused for a given time limit.

Swapping schedulers can be used to drive many application sys- tems. This is particularly beneficial when related systems are present. For example, an installation may have systems for in- ventory, order entry, receivables, payables, and ledger. If the systems are broken down into specific stages (or vectors), a single order entry transaction can be transferred through inven- tory (to adjust items ordered), order entry (to print a shipping notice), receivables (for a new bill), and ledger (to post the cash flows between accounts). Furthermore,

the same transaction data can be sent to report stages for queueing daily summary information.

Summary

Many applications of process handling have been examined, some of which are very appealing. A good process handling manager can provide control and communication which are vital for comprehen- sive systems. However, careful consideration should be given to the complexity of the system, limitations of the computer, and error handling facilities. Complex process handling systems pass around a great deal of information, and the possibilities of sys- tem failure, program abort, invalid data, and program detected errors must be taken into account. For example, what does a stage do if an IMAGE error is encountered? What does the scheduler do if a stage aborts?

As mentioned in the introduction, one may swear by or swear at process handling systems. The division is largely up to the process handling manager or scheduler in large systems, but also relates to the resources available in the machine. There are few, if any, instances which require process handling. However, there are instances which may benefit from process handling. In the hands of a careful analyst, it can prove itself to be an in- valuable tool in the design of powerful, efficient transaction processing systems.

Brief Biographical Sketch of Jeffrey R. Kell

Jeff Kell, 25, has been in data processing for eight years, with a background in systems programming on IBM equipment for three years, and five years with Hewlett-Packard equipment. Jeff has been with the University of Tennessee at Chattanooga for six years, and serves as senior systems analyst for Administrative Computing Services. In addition, he maintains configurations, backup, and recovery for the university's five HP-3000 computer systems.
