IMAGE threads investigation report

» Return to original page

TurboIMAGE thread-aware and thread-safe investigation report

by Tien-You Chen

1. Introduction

1.1 Background

In order to respond to users' requests quickly, this year, 2001, Interex and CSY have teamed up and taken a different approach to handling the System Improvement Ballot (SIB). First, each Special Interest Group (SIG) proposed a list of enhancements, less than 10 items, to a

specially formed committee. Second, the committee went through the lists from all the SIGs and picked 20 to 30 enhancements and created a final ballot. After all the users of HP e3000 community have voted, CSY will quickly review the result and choose several items from the top ten and then begin working. The item, which got the most votes, was "make TurboIMAGE thread aware and thread safe".

1.2 Requests

The details of the item, "make TurboIMAGE thread aware and thread safe", actually contain many aspects. This item is combined of a couple of enhancement requests from the SIGIMAGE enhancement list. One request is "make TurboIMAGE thread aware and thread safe"; the other request is "allow a process to pass a DBOPEN to a son process if the database is in stable status". There is one more aspect of this item, which is "pass ‘base ID’ between processes". People hope one solution can solve as many issues as possible. Even though these components have the same concept, which is trying to share DBOPEN between processes, solutions to these components are quite different. From the view of the dispatcher of a system, thread, son process, forked process and normal process are the same, i.e. these are all processes and have unique process numbers (PIN). However, threads, forked processes, son processes and normal processes are different in many ways. The major difference is thread shares the SR5 space with the initial thread which creates it, but the forked process has its own SR5 space; the forked process inherits opened files from its parent process but a son process created by ‘createprocess’ does not. Because of these differences, leveraging the code from one component to another may be minimal.

This document lists issues for each component and the possible solutions. Based on the resources availability and the customers’ need, we may choose to implement whole or just a portion of this item.

1.3 Acknowledgements

I would like to thank the following people for sharing their MPE, thread knowledge with me:

Bill Cadier - for his clear and detailed paper, Pthreads for the HP e3000

Craig Fairchild - for the internal of the file system he exposed to me

Marcia McConnell - for her insight on the thread implementation in MPE/iX

2. Problem statement

2.1 Thread safe

In the traditional MPE environment, every execution of a piece of code is called a process. The definition of a process is "the unique execution of a particular program at a particular time". A unique number (PIN) is assigned to a process, coupled with a stack, heap space and many data structures in the system to form an executable identity. Since MPE moved toward open systems, many features have being added to MPE; POSIX Thread is one of these. Thread is somewhat like a process, but uses less resource and needs less work to create and destroy. A thread has its own PIN and has its own data structures such as PIB, PIBX, PCB and PCBX. A thread has its own stack but shares the same SR5 virtual space with the initial thread, which creates it. A thread also shares the open files with its initial thread. Because of these special characteristics, TurboIMAGE database access can not be handled correctly in the thread environment. The reason why TurboIMAGE is not safe in thread environment can be divided into the following:

2.1.1 Runtime control blocks

For every DBOPEN user's program called, a runtime control block named DBU will be created. A DBU contains the information specific to this DBOPEN. This includes:

- transaction information (XMUI, GTX'id, 00-file…etc)
- locking information (data set and/or predicate level locks it owns, pointers to owned locks…etc)
- logging information (transaction id, MDBX base ids, log buffer)
- current record pointers as well as back/forward record pointers for each data set
- open data set files and SMCB of those files
- current item lists for each data set
- critical level, system depth
- error handling
- dbcontrol settings for this OPEN
- fields for bookkeeping purpose (intrinsics count)

DBU also has a trailer area to be used as a temporary data processing space. Each DBOPEN has one DBU. If a process opens the same database twice or a process opens two databases, then two DBUs will be created. A process can have maximum of 127 DBUs created.

A DBUX runtime control block will be created when the process first opens a database. Only one DBUX exists for each process and that contains:

- pointers to all DBUs which belongs to this process
- TPI/TPS Plabels
- lock table pointer
- GTX table pointer
- dynamic rollback record number mapping table for rollback activity
- process termination trigger

Both DBU and DBUX reside in SR5 virtual space. Since threads share the SR5 virtual space, it is natural to think we should share the DBU and DBUX as well. However, both DBU and DBUX are designed for not sharing between processes (in other words, shared by different PINs). It is assumed that only the process, which creates the DBU and DBUX, will access it. So there is no locking mechanism to protect the write to fields in DBU and DBUX from other processes. And it is not safe (incorrect) to share the XM logging, data set and predicate level locks. Current record pointers will be messed up if more than one thread updates the pointer for the same data set. Critical level, intrinsic count and DBCONTROL settings may all incorrectly reflect the real value and lead to data corruption or even a system abort.

There are other runtime control blocks been used during database access, such as DBG, DBB and DBS. DBG and DBB are created when the first user opens a database. One DBG and one DBB are created for each database opened and are shared by all the users on the system. Only one DBS is created on a system and is also shared by all the users. All of them are originally designed for sharing among the users, so they should be safe for thread accessing. However, there are still a few things that need to be taken care of. For example, an "accessor entry" will be allocated in DBG for every DBOPEN. The entry is identified by a PIN and the open count, and has a pointer pointing to the DBU that was created by this DBOPEN. If more than one thread shares the DBU, should each thread have its own 'accessor' entry?

2.1.2 Global variables

TurboIMAGE is composed of a set of procedures called 'intrinsics'. Since thread has it own stack, procedure local variables, which are allocated when calling TurboIMAGE intrinsics and deallocated when exiting the intrinsic, are not problems in thread environment. However, there

are a few global variables exist, since they have to last and keep the value from one intrinsic call to another intrinsic call. Those variables are created by TurboIMAGE and allocated in the DP area in the SR5 virtual space. These variables are:

- qlock_trace: used to store debugging and tracing flags set by the user
- bti_globals: b-tree related flags, the timestamp… etc
- ccu_globals: jumbo data set related flags, attach/detach plabels, the timestamp …etc
- chunk_ctl: one object for every jumbo data set opened by the process, which contains file related information (chunk data files) such as file ids, file SMCBs, open options and the error status

These global variables should be taken care of in the thread environment.

2.1.3 Open files

A thread inherits the file structures of its initial thread; in other words, both threads share the same open files. Thread programmers can call wrapped functions instead of normal intrinsics in order to allow the file system to protect the data structures from multithreaded access. For example, the thread program may call thd_fopen instead of fopen to open a file. That should work fine for thread calling file system intrinsics, but how about TurboIMAGE database access?

A database is a collection of files; database access actually ends up with file access. However, only the root file is accessed by TurboIMAGE using file system intrinsics. For the user data, TurboIMAGE uses lower level operating system procedures i.e. disc storage management routines, to handle the reading and writing of data. Disc storage management routines then call transaction management (XM) routines to ensure the physical and/or logical data integrity. Sharing opened files means sharing the SMCBs (Storage Management Control Block), so TurboIMAGE must introduce a mutex-like locking mechanism to lock out other threads from accessing the same SMCB if one thread is reading or writing the same file.

2.2 Forked processes

In addition to thread, forking is also a new feature, which was introduced while MPE moved toward open systems. A forked process inherits many characteristics from its parent process, including open files. A forked process has its own data structures such as PIB, PIBX, PBC and PBCX as well as its own SR5 virtual space. However, if a process already opened one or more databases while forking, the forking step will fail. That is because DBU and DBUX are open files from the file system's point of view, but DBU and DBUX are new, no-named and privileged files that the file system does not currently handle. So, how can we make forking work with TurboIMAGE? Again, let's assess the problem according to the following three points:

2.2.1 Runtime control blocks

DBU and DBUX are the main obstacles while forking. The file system duplicates many data structures of open files for a forked process. So, is it that simple just to ask the file system also to copy DBU and DBUX while forking? No. Just duplicating the open DBU and DBUX and copying the content does not solve the problem, because the data inside the DBU/DBUX includes many pointers. They include pointers to each other (DBU to DBUX and DBUX to DBU), pointers to DBG and DBB, pointers to locked entries in DBG and pointers to data sets. So, simply copying the data in DBU will result in meaningless data.

The state of the forking process is another thing has to be taken care of. The forking process

can not be within a transaction while forking, either a dynamic transaction or a user logging transaction. The forking process can not own any locks, no matter if it is a database lock, a data set lock or a predicate lock. And, of course, the forking process can not be in a TurboIMAGE intrinsic at the time of forking. If the forking process is enabled for user logging, even when forking happens between two transactions, one more step needs to be taken: a fake DBOPEN log record must be written into the log file for the forked process. Otherwise, DBRECOV will fail to recover the transactions for the forked process.

2.2.2 Global variables

Since the forked process has its own SR5 virtual space, DP-relative global variables should not be a problem. Somehow, TurboIMAGE needs to find a way to copy the DP-relative global variables from the forking process to the forked process.

2.2.3 Open files

The forked process inherits the file structures of its parent process by having it own PLFD (Process Local File Descriptor) but sharing the same GDPD (Global Data Pointer Descriptor). Because it shares the GDPD, it shares the SMCB inside the GDPD. Because it shares SMCB, it has the same issue as threads have.

2.3 Passing "base ID" between processes

The third request in the same arena is passing "base ID" between processes. The processes may mean threads, or may mean the forking and forked processes, or parent and son processes, or even two unrelated processes. Before a discussion of this problem, let's first understand what the "base ID" is. "Base ID" is a unique number (unique to this process), which TurboIMAGE returns when a program calls DBOPEN. The program passes a database name to DBOPEN with the first two-byte set blank. TurboIMAGE, after successfully opening the database, will put the "base ID" in this two bytes space. Basically, the "base ID" is the index to an array of pointers, which point to DBUs this process has created. This array is located in the DBUX control block. For every DBOPEN, TurboIMAGE creates a DBU control block in the SR5 virtual space. The address of the DBU is then added to the array in the DBUX, which belongs to this process. So, by getting the "base ID" from all the subsequent TurboIMAGE intrinsic calls, TurboIMAGE can link the intrinsic call to the related DBOPEN and the DBU control block. So, passing "base ID" between processes actually means sharing the DBU.

Since both DBU and DBUX are considered to be "process local", sharing DBU between processes is really deflecting the architecture. Many issues will surface by doing that:

- What if the process, which receives the "base ID" already has an open database? The "base ID", which really is an index, may mess up with the one it already has.
- Who will be the "owner" of the DBU? What if the ID has been passed through several hands?
- What should the process termination do, if the program is aborting, the DBU is being purged, but the other process is accessing the DBU?
- How should the user logging be handled? For example, how can DBRECOV handle all the TurboIMAGE intrinsics without matching DBOPEN?

3. Options

After gone through various issues for accessing databases via threads or forked processes, let's discuss the options for solving these issues. Again, let's divide them into three different

categories: runtime control blocks, global variables and open files.

3.1 Runtime control blocks

There are two ways to handle the runtime control blocks: copy and share.

3.1.1 Copy DBU and DBUX

When a process, which already has an open TurboIMAGE database, forks, one way to handle the DBU and DBUX runtime control block is to copy. Copy all the DBUs and the DBUX, which the parent process owned, to the SR5 virtual space of the forked process. The steps to copying should be pretty similar to the steps in the DBOPEN, i.e. create DBUX, create DBU, add DBU address into DBUX, increment the database open count, then add "accessor entry" into DBG and link to DBU. The difference is, the copying step needs to create all the DBUs, which the forking process have. Additionally, it should also copy the contents in the DBUs to newly created DBUs because the forked process should inherit whatever the forking process has. So, except for the transaction and locking related fields, the following items should all be copied to the newly created DBUs: the current record pointers of each data set, open data set files, current item lists for each data set as well as DBCONTROL settings. There are advantages and disadvantages to this approach:

Pros:

- Current record pointers will not mess up. For example, the forking process reads down a chain and then calls fork. If the forked process shares the DBU and does a DBFIND to the same data set as the forking process, then the current record pointer will be reset to the beginning of the chain and will cause the forking process to get a wrong record for the next chain read. Copy DBU will avoid this kind of problem.

- Several fields in DBU are used to store the current XM status and transaction ID. Some other fields are used for the locking status and for pointing to locks, which the process owned. Separating the DBU can clearly distinguish who owns what. Better yet, each thread or process can have its own locks or transaction at the same time.

- For each DBOPEN, TurboIMAGE increases the user count in the root file as well as in DBG. TurboIMAGE also creates an "accessor entry" in DBG with the owner PIN and the address of the DBU, which is created by this DBOPEN. Copy DBU(s) then create "accessor entry" for each DBU. This is a straightforward thing to do while forking.

- Each DBU has a trailer area, which is a scratch space for each TurboIMAGE intrinsic to process intermediary data. For example, the space is used for converting a data set name to a data set number, processing an item list array or constructing a full record of data if the user only passed in a partial record. If each process has its own DBU, then we do not need to develop a complex trailer-area management procedure.

- The process termination procedure involves releasing all the locks the process still owns, purging DBU and decreasing the open count from DBG and the root file. If each thread or process has its own DBU, DBUX and "accessor entry", then the process termination steps will be the same as usual. Otherwise, DBU/DBUX needs to keep track of the number of processes accessing the control block and it needs to act accordingly.

Cons:

- The effort to duplicate DBU(s) will be similar to a lightweight DBOPEN. Since the DBOPEN intrinsic already has been criticized as a single threaded, time-consuming

procedure, the performance of the forking will be bad if we choose to copy the DBU(s).

- Threads share the same SR5 virtual space. So, if the initial thread opens many databases and then creates many threads, then there may be a space problem if we copy DBU(s) for every thread.

- One general concept for thread and forked processes is that they inherit (share) the resource of the parent process. For instance, threads share the open files; thus they also share the current record pointers. So, conceptually, threads should share the current record pointers for every data set. If TurboIMAGE copies DBU, that will break the convention.

3.1.2 Share DBU and DBUX

Instead of copying DBU and DBUX, it makes more sense for thread to share DBU and DBUX. However, since the DBU and DBUX are designed to be accessed only by one process, sharing methods need to be added to handle the multi-accessing of a DBU. For example, DBU used to be owned by a process, now an array of accessing PINs may be needed in the DBU. We may also need several semaphores to control the write to many tables in DBU. The trailer area needs to be divided into sub-areas. So, different threads can work on the same DBU at the same time. If we do not divide it, one thread may lock out the other threads for the whole duration of an intrinsic, then the concurrency will be an issue. Let's also list the pros and cons of this approach:

Pros:

- To share the resources with the initial thread is the key point for the thread environment; especially all threads share the SR5 space. So, intuitively, all threads should share DBU and DBUX.

- A table in DBU keeps all the open file information, that includes file id, file offset and SMCB virtual address …etc. One entry exists for each open data set. Since threads share the files, it is natural that they share this table in DBU.

- The user local data set table in DBU holds current record pointers for each data set. Share DBU means sharing the record pointers for every data set. It is the convention for files, but how about data sets? Which way make more sense to the user? Which way can benefit users more?

Cons:

- All the transaction-related information is stored in DBU. In order to share the DBU among threads, some kind of lock needs to be used to protect the on-going transaction. User may or may not explicitly start a transaction. If the user does not explicitly start a transaction, in order to maintain data integrity, TurboIMAGE will create a transaction internally for each modification intrinsic. So, the duration of a transaction can be as short as a TurboIMAGE intrinsic or as long as hundreds or thousands of intrinsics. In other words, the duration to hold the lock can be too long to be practical. The same concern also applies to those fields in DBU for process-owned locks. Maybe we should allocate different spaces for different threads.

- The DBU trailer area is another big concern when it is shared by threads. The trailer area is continuously being used over the whole duration of processing an intrinsic. The performance concern would lead us to design a more sophisticated space management system instead of a single threaded mechanism.

- The log record buffer is located in the DBU. That is another big chunk of space that can not be shared by threads. Apparently, the DBU size should be increased to a

certain level in order to accommodate the multiple copies of the log record, trailer area …etc.

- For every DBOPEN, one "accessor entry" and one DBU will be created. The address of the DBU will be stored in "accessor entry" and the user count will be increased. These three are tightly coupled to each other. If we share DBU with threads, should we share the "accessor entry" or create a new entry for each thread? Should the user count match the number of DBUs or the number of threads? No matter what choice we make, the meaning will be changed.

3.2 Global variables

Compared to runtime control blocks, the issue of global variables is trivial. Global variables reside in the SR5 virtual space. Because of the nature of forking a process versus creating a thread, global variables might be shared by threads and should be copied for forked processes.

3.3 Open files

Both thread and forked process share open files with their parent process. Because of that, parent and child processes share the same pointers of those open files. Even though all the data sets are files, TurboIMAGE, instead of using the current record pointer of the file system, maintains the current record pointer internally for each data set opened and for every DBOPEN. This is because the definition of "record" is different in the file system and in TurboIMAGE. In the case of "serial read" or "chain read", TurboIMAGE figures out the next record number to fetch, then calculates the offset in the data set file. After filling in the fields in SMCB, TurboIMAGE calls the disk storage management routines to retrieve the data.

In DBU, TurboIMAGE has two tables: the User Local Data Set Table maintains the current record pointers and backward/forward record pointers for each data set and the Data Set File Table keeps the open file information for every data set. For the processes to share open files actually means sharing those tables in DBU. It makes sense to share the SMCB so child processes do not need to open the data set again. However, I wonder how much benefit users can get if TurboIMAGE allows sharing record pointers between processes.

4. Feedback

Before I finalize this document, I sent out a preliminary investigation report to many experts in the HP e3000 community asked for their feedback and opinions. Here is the summary of their suggestions:

Even though to make TurboIMAGE thread aware and thread safe is the main topic, to allow forking work with TurboIMAGE is also important. Several people think the fact, if a process already have a database opened will cause fork() to fail, is a bug which we should fix.

Concurrency and performance are the essence of the thread. If we choose to share DBU and DBUX by the threads; but for the integrity reason, we lock out each other to access the DBU for the duration of processing the intrinsic or even whole transaction; that will be unacceptable. In other words, we have to develop a sophisticated method to control the access to DBU in the thread environment so that each thread can act like an independent process and concurrently accessing the database.

If a thread programmer programs more than one thread to access the same file, s/he is aware that those threads are sharing the same record pointer. So, it is also the programmer's

responsibility to keep track of current record pointers if more than one thread would access the same data set in the same database.

Many design suggestions such as adding new intrinsics for thread operation to use; implementing "internal MR" to prevent potential deadlock between two PINs (but in the same "thread family"); creating "light-weight" DBU, which contains only things need to be separated etc. are also mentioned in their feedback.

5. **Recommendation and estimation**

Based on the above discussion, I believe "pass base ID between processes" is not as important as the other two requests. It is too ambitious to conquer this generalized enhancement, which manipulate the base ID between any two processes; it is also not clear that the benefit to the customer would be. Despite the complexity and performance issues, the logical conclusion is "sharing for threads" and "copying for forked processes". Because threads share the SR5 space, they could also share DBU/DBUX and global variables. And because forked processes have their own SR5 spaces that make more sense to copy the DBUX/DBU(s) and global variables from the forking process. The implementations will be different for these two different approaches. Based on the resources we currently have, I do not think we can address them both at the same time. To implement in stages would be the right approach.

6. **Issues**

There are also several issues lie in front us if we do decide to implement either or both of the enhancements.

Right now the performance is already a big issue for fork() on HP e3000, especially compared with UNIX fork(). We can spend a lot of effort to make TurboIMAGE work with fork(), but if the overall performance is not acceptable, it is in vain. Worse than that, the additional work for opened database(s) to fork is not simple. First we need to check all the databases, which have being opened by the forking process, to make sure it is in the right state to fork. Then we copy all the DBUs and DBUX the forking process has into the SR5 space of the forked process. The third step is to initialize the fields in DBU/DBUX that is not suitable for duplication. And the last step is to link/update the database open information in the global data structure. The whole procedure is similar to a "light-weight" dbopen, which is pretty time consuming. So, the performance of the overall forking procedure may get worse then current.

» Return to original page