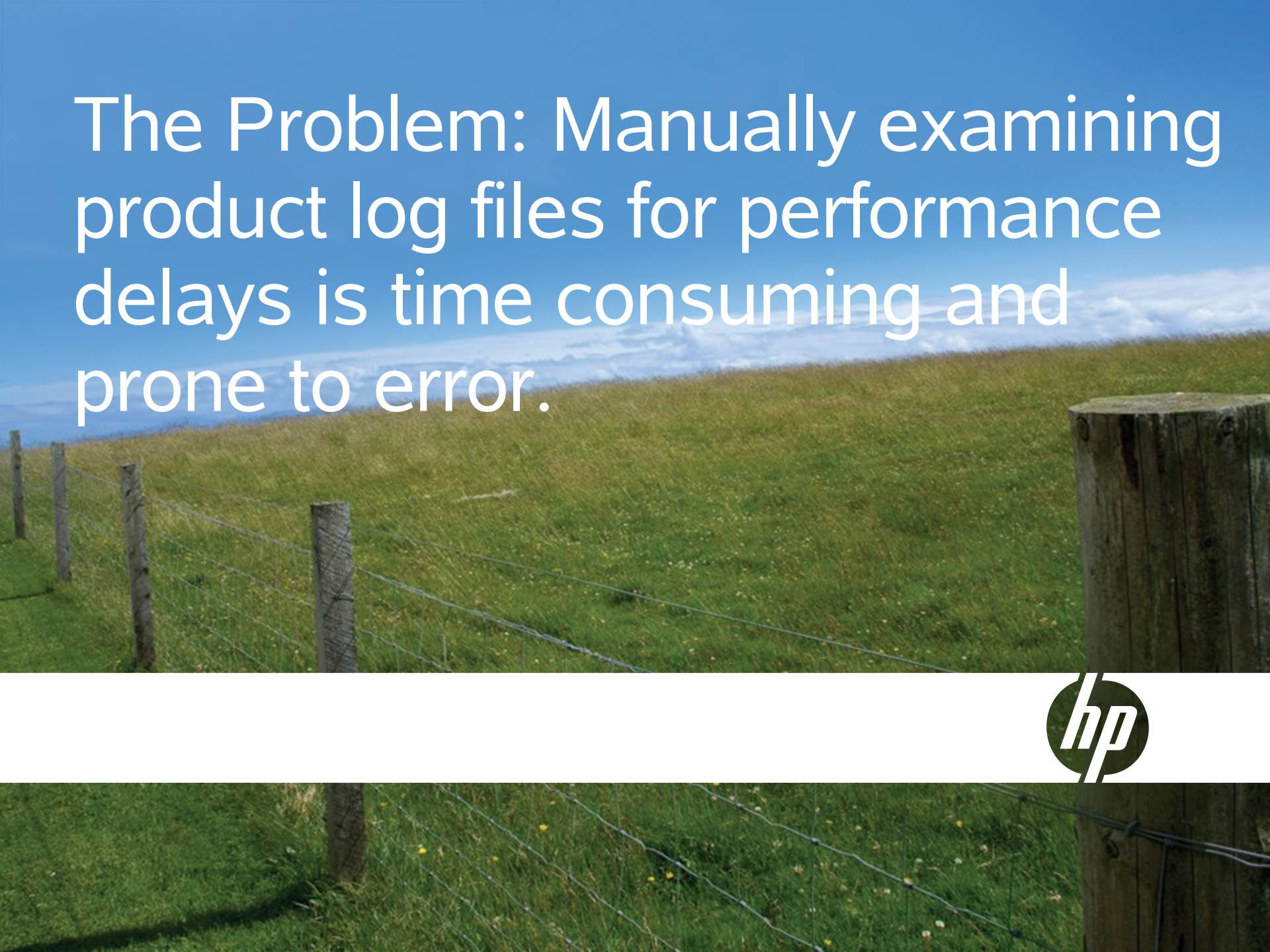
Scripting Performance Troubleshooting using Samba Debug Logs

Don McCall

Master Technologist, WTEC HP


The Problem: Manually examining product log files for performance delays is time consuming and prone to error.

One Solution: Perl Scripts to isolate 'significant' lines in debug files.

WTEC_Support_Tools

- spotdelay.cifs.server
- spotdelay.tusc
- analyze.cifs.server
- analyze.tusc
- fileopens.cifs.server
- getinfo.cifs -s

spotdelay.cifs.server

- Logging prerequisites
  - `Log file = /var/opt/samba/log.%m`
  - Log level = 6
  - Debug hires timestamp = yes
- Spotdelay.cifs.server -400 <log.machinename>
  - would report any log lines that were written more than 400 milliseconds after the previously logged entry, along with the line number of the log file where the delay occurred.
  - You could then examine the log file at and around this entry to determine what was happening that might have caused this delay.

spotdelay.tusc

- Logging prerequisites
  - tusc –aeflpT <smbd pid>
  - Tusc is HP-UX only; but similar to strace on Linux systems; spotdelay.tusc could easily be adapted to work against strace output.

•spotdelay.tusc -2 <tusc.output.file>

  - scan the tusc.output.file for lines with a delay greater than 2 seconds.

•spotdelay.tusc –n .002 –s fcntl

  - Display lines from tusc output with diff of .002 seconds (20 milliseconds) specifically involving the fcntl() system call.

spotdelay.strace

- Logging prerequisites
  - Strace -fttTo <straceoutputfile> -p <smbd pid>

•spotdelay.strace -2 <straceoutputfile

  - scan the straceoutputfile for syscalls with a delay greater than 2 seconds.

•spotdelay.strace –n .002 –s fcntl <straceoutputfile

  - Display lines from strace output with diff of .002 seconds (20 milliseconds) specifically involving the fcntl() system call.

analyze.cifs.server

- Logging prerequisites
  - Log file = /var/opt/samba/log.%m
  - Log level = 6 (or higher)
  - Debug hires timestamp = yes
- For each type of cifs request/response pair, generates info such as
  - Total time subsumed
  - Avg time per pair
  - Max/Min time in pair
  - Number of pairs
- Also gives these stats overall for all cifs request/response pairs.

analyze.cifs.server (continued)

Analyze.cifs.server Example:

OUTPUT REPORT:
==========

SMBdskattr:
- Total time consumed: 0.000834999998915009
- Average time consumed: 0.000834999998915009
- Minimum time consumed: 0.000834999998915009
- Maximum time consumed: 0.000834999998915009
- Number of times called: 1

SMBecho:
- Total time consumed: 0.000465000004624017
- Average time consumed: 0.000465000004624017
- Minimum time consumed: 0.000465000004624017
- Maximum time consumed: 0.000465000004624017
- Number of times called: 1

SMBtconX:
- Total time consumed: 0.00200600000243867
- Average time consumed: 0.00200600000243867
- Minimum time consumed: 0.00200600000243867
- Maximum time consumed: 0.00200600000243867
- Number of times called: 1

SMBtdis:
- Total time consumed: 0.000834999998915009
- Minimum time consumed: 0.00200600000243867
- Maximum time consumed: 0.00200600000243867
- Number of times called: 1

SMBtrans2:
- Total time consumed: 0.00302799999917625
- Average time consumed: 0.00302799999917625
- Minimum time consumed: 0.00302799999917625
- Maximum time consumed: 0.00302799999917625
- Number of times called: 1

Summary:
- Lines processed: 639
- Unique CIFS requests: 5
- Most called CIFS request: SMBdskattr
- CIFS Request using the most time: SMBtrans2
- Total CIFS requests made: 5
- CIFS Log Duration(seconds): 2.9692040000009

analyze.tusc

- Same logging prerequisites as spotdelay.tusc
- NOTE strace on many linux distros already natively contains functionality to report on time spent/%time spent per system call, # system calls, etc. So analyze.tusc probably not worth porting across to a strace capable distro.

fileopens.cifs.server

- Logging prerequisites
  - `Log file = /var/opt/samba/log.%m`
  - Log level = 10
  - Debug hires timestamp = yes (optional)
- Script Output
  - fnum=13460 fname=us_dc/mcpolylayer4/lmaux.btr write=No opened=[2007/12/07 10:34:50, 5] at 439457 closed=[2007/12/07 10:34:50, 5] at 439995 open for 0 seconds

getinfo.cifs -s

- Not a performance script per se
- Collects background/environment information on a system running Samba

  getinfo.cifs -t S|C|P -p outputpath -o tarfilename -X -x -lc -lh

  -t = S(server)|C(client)|P(pam_ntlm)

  -p = path to create collectiondata and tar file(default = /tmp)

  -o = tarfile name to create(default= CIFS.tar)

  -x = collect sensitive data (/etc/passwd,smbpasswd,etc

  -X = do not collect HPUX config files

  -lc = collect CIFS log files

  -lh = collect HPUX log files

  Defaults: -t S:-p /tmp:-o cifs_diag.tar: -x off:-X off:-lc on:-lh on

A brief look at the fileopens.cifs.server script

fileopens.cifs.server sections

- Setup
- Main processing loop
- Filter processing
- Time calculation

Setup

- Variables and filters defined here based on specific log file familiarity:
  - my ($timeentry_grepper) = '^\[\d\d\d\d';
  - my ($key_grepper) = 'allocated file structure';
  - my ($starttime_grepper) = 'reply_ntcreate_and_X:';
  - my ($endtime_grepper) = 'freed files structure';
  - my ($auxinfo_grepper) = 'opened file';

Filter processing

- The `process_filter` routine takes the following parameters:
  - a search string
  - value associative array
  - time associative array
  - line number associative array
  - index value or offset (depends on last passed parameter)
  - current line position (base 0) for associative array value
  - "value" or "index" determines whether index value or index position is passed into the routine.

Filter processing

```
sub process_filter {
        $line_to_process = "";
        if( /$_[0]/ ){
        print "process_filter: _[6] = -$_[6]-\n" if $debug;
        @line_to_process = split;
        if($_[6] =~ "I"){
        print "process_filter: its an index!\n" if $debug;
        $pindex = $line_to_process[$_[4]];
        } else {
        $pindex = $_[4];
        }
        $_[1]{$pindex} = $line_to_process[$_[5]];
        $_[2]{$pindex} = $timeline[0] . "]" ;
        $_[3]{$pindex} = $.;
        print "process_filter: key=$pindex, a1{key}=$_[1]{$pindex},a2{key}=$_[2]{$pindex},a3{key}=$_[3]{$pindex}\n" if $debug;
                }
}
```

Time calculation

- Calctime() routine takes two parameters:
  - Starttime
  - Endtime
- Returns the difference between the two (in seconds)

Time Calculation

```
sub calctime {

        ($opentime,$closetime) = @_;
        local($discard, $opened, $discard2) = split / /,$opentime;
        local($discard, $closed, $discard2) = split / /,$closetime;
                local($hour, $minutes, $seconds) = split ":", $opened;
                local($calcopentime) = ($hour*3600) + ($minutes*60) + $seconds;
                local($hour, $minutes, $seconds) = split ":", $closed;
                $calcclosetime = ($hour*3600)+($minutes*60)+$seconds;
                print "calctime: open time: $calcopentime\n" if $debug;
                print "calctime: close time: $calcclosetime\n" if $debug;
                $diff = $calcclosetime - $calcopentime;
                print "calctime: Diff=$diff\n" if $debug;
          return $diff;
}
```

Main processing loop

- Define the associative arrays we will stuff our data into
- Read the log file, line by line, searching for lines containing our key, and the various data we want to associate with the key
- Use the process_filter() routine to process the lines found to associate the appropriate data valueswith the appropriate key value
- Sort and print the arrays built in HRF (human readable format) or CSV (comma separated values)

Main Loop

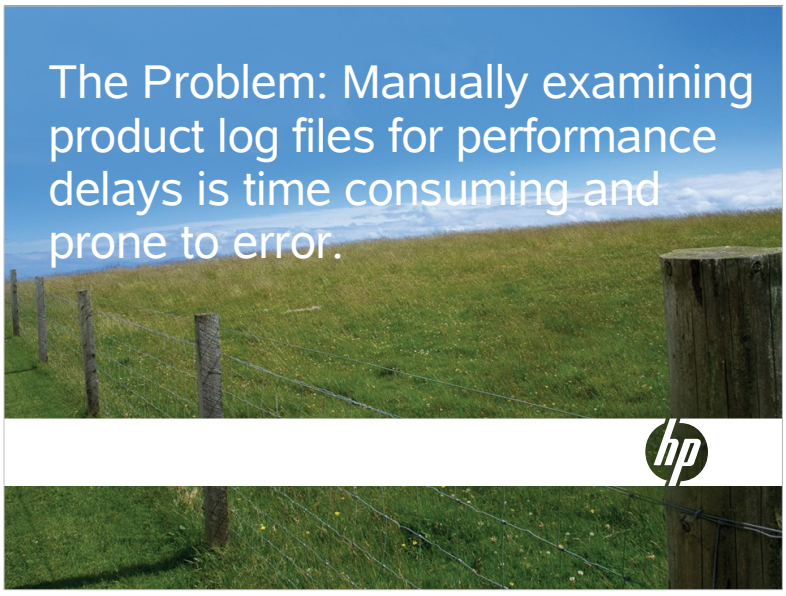
```
# MAIN loop:
while (<>) {
        chomp;
        if ( /$timeentry_grepper/ ){
                @timeline = split(/\]/,$_);
                next;
                } else {
# get fnum for this file
        if ( /$key_grepper/ ){
        @temp = split;
        $current_fnum=$temp[6];
        print "current_fnum = $current_fnum\n" if $debug;
        }
#get the open info for this file:
        process_filter($starttime_grepper,*opens,*opened_time,*opened_line,$current_fnum,7,V);
# get the read and write access.  NOTE if a directory, this will not be
# available.  This will result in a readwrites{} of null for the current_fnum.
        process_filter($auxinfo_grepper,*readwrites,*readwrites_time,*readwrites_line,$current_fnum,5,V);
# now process the close and get the closed time and closed line:
        process_filter($endtime_grepper,*closes,*closed_time,*closed_line,3,4,I);
        }
}
#Print out the results in HRF (human readable format)
        foreach $key (sort keys(%opens)) {
        print "fnum=$key fname=$opens{$key} ";
        if($readwrites{$key}) {
                print $readwrites{$key};
                 }else{
                print "DIR";
                }
        print " opened=$opened_time{$key} at $opened_line{$key}";
        print " closed=$closed_time{$key} at $closed_line{$key}";
        print " open for " . calctime($opened_time{$key},$closed_time{$key}) . " seconds\n";
        }
```

Where to get the scripts?

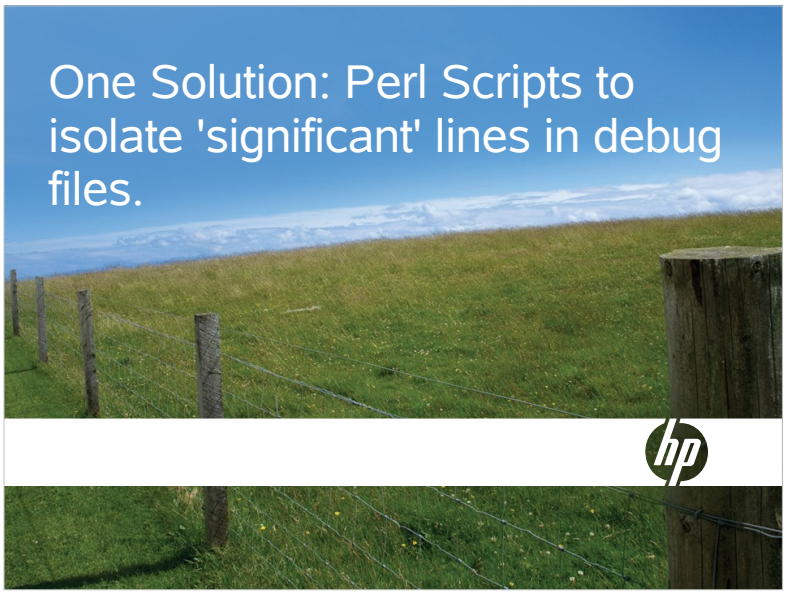
- If you have an A.02.02 or greater installation of HP CIFS Server (Samba), the tools and man pages can be had from the following directory:
  - opt/samba/WTEC_Support_Tools
- Download the HP CIFS Server product from
  - http://software.hp.com
    - 'Internet ready and networking
      - HP CIFS Server
  - This will give you a .depot file, which is basically just a tar file; extract the WTEC_Support_Tools directory via
    - Tar -xvf HPUXxxxxxxxx.depot CIFS-Server/CIFS-UTIL/opt/samba/WTEC_Support_Tools

Thanks for your Time!

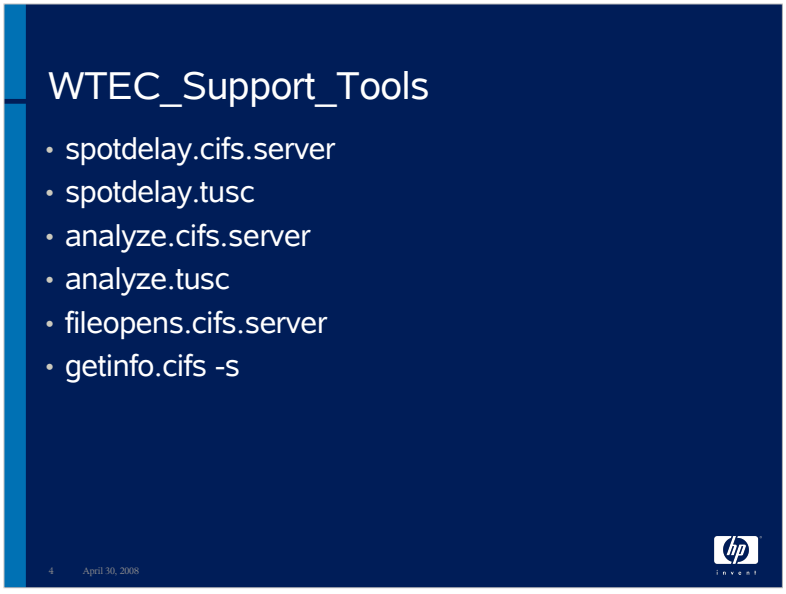
- I'm Don McCall, and I work for a 3rd level support organization called WTEC (Worldwide Technical Expert Center) at HP. This October will mark my 27th anniversary with HP, and I've been involved in Samba and Samba-like (think Advanced Server for Un*x, or LanManager for Un*x) product support for the last 14 years or so. We support 1st and 2nd level organizations in HP to get problem resolution (proactive and reactive) issues addressed for HP internal and external customers.
- Along the way, we generate a lot of 'one-off' tools to parse data, and address data collection issues during problem solving situations. Sometimes these can be generalized to fit a wider area, and when they are, we work with our lab to get them included in the product distribution that we ship with the HP-UX OS. The tools I am going to talk about today evolved in just this manner, and it is my hope that those of you that work with HP-UX and Samba can get some use out of them as they are. For those of you that work with Samba on other platforms, like SUSE, Red Hat, Unbuntu, Solaris, etc, I present these scripts as a starting place for your own scripting development, if you find them useful.



- The impetus for creating this paper and the tools it describes was the HP CIFS product suite, which includes Open source Samba as the CIFS Server component. These applications are capable of creating very detailed log files complete with timestamps. Since they are Client/Server applications, it is possible that a performance problem could express itself in one or more of several areas: application code, interface to system resources (system library calls), h/w subsystem latency (e.g.: disk, memory cache, or network buffer resources), or network issues (retries, poor routing, slow network, etc). Information in these log files could implicate any of these areas, but it is tedious to sift through the log files looking for significant delays.
- In addition, with debug entries possibly numbering in the tens of thousands, it is easy to miss something important.
- Finally, these log files cycle at some configurable point, so it is possible that the debug file, if not captured in a timely fashion, might miss the performance inhibiting event. Having a simple one step method for customers to check a debug file for delays before sending it to HP for analysis is critical to avoid iterating through a 'capture/send to HP/analyze/recapture' cycle that introduces unnecessary delays in the problem solving process.
- It's important to note that these tools and techniques in no way replace a good performance measurement suite. They were developed to use in a problem solving environment where you may have little else available on the target system to work with, as opposed to a performance testing environment where you have full control of the system, load and applications running there.



- I need to preface this whole talk with the statement 'This is NOT rocket science.' They're simple Perl scripts.
- To address this issue, we have created a suite of Perl scripts that are designed to act upon either a HP CIFS product log file, or a tusc1 trace of an HP CIFS process, and output points of interest in the log files that require further examination from a performance standpoint. Perl was chosen for the script language because of its wide acceptance/delivery on HP-UX OS distributions.

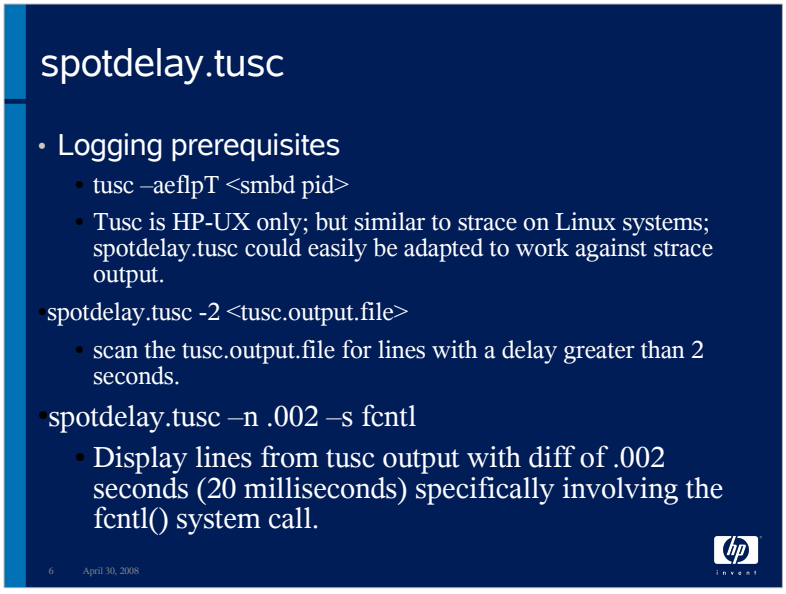


- I will only be talking about a subset of the tools we ship in the WTEC_Support_Tools directory, as some of them are specific to the CIFS Client. But these six I think are quite useful.
- All of these are available free with the HP CIFS Server download from the HP web site, and are published under the GPL license version 2. But they ARE written and tested only on the 3.0.22 Samba code base, as this is the code base we use in building the HP CIFS Server version. In addition, the paths in some of the scripts are HP specific. So if you plan on using these on a Linux distribution, there will be some minor modifications you will need to carry out. In addition, all of these scripts depend on certain patterns existing in the debug statements they use to key off of. So later Samba code bases may require some tweaking to the patterns used in the scripts, in the event that debug statements have changed.
- Finally, I include two scripts that actually work off of 'tusc' output files – tusc is similar to strace in many Linux distros, but in some ways is not as powerful. A lot of what analyze.tusc does, for instance, is already native to strace, so it's probably not worth porting over to use on a Linux distribution.

spotdelay.cifs.server

- Logging prerequisites
  - `Log file = /var/opt/samba/log.%m`
  - Log level = 6
  - Debug hires timestamp = yes
- Spotdelay.cifs.server -400 <log.machinename>
  - would report any log lines that were written more than 400 milliseconds after the previously logged entry, along with the line number of the log file where the delay occurred.
  - You could then examine the log file at and around this entry to determine what was happening that might have caused this delay.

- This script examines an HP CIFS Server debug file and reports line numbers in the file that have a delay greater than a specified number of milliseconds. It also will report a delay if the specified millisecond count is exceeded within 10 lines in the file. The output from this script gives specific line numbers and times to examine in the log file, significantly reducing the time needed to pinpoint and locate potential contributing factors to a performance problem. Examining the log statements immediately prior to the reported line numbers may help narrow down the possibilities for performance inhibitors
- For instance, you might see a log line that indicates that a server response was put out on the wire, but the next client request doesn't show up for some time, possibly indicating some unexpected processing or delay on the client side to be investigated. Or a name to uid lookup where the response takes an abnormally long time to fulfill might send you off investigating the robustness of the name service switch (NSS) backend being used underneath Samba....



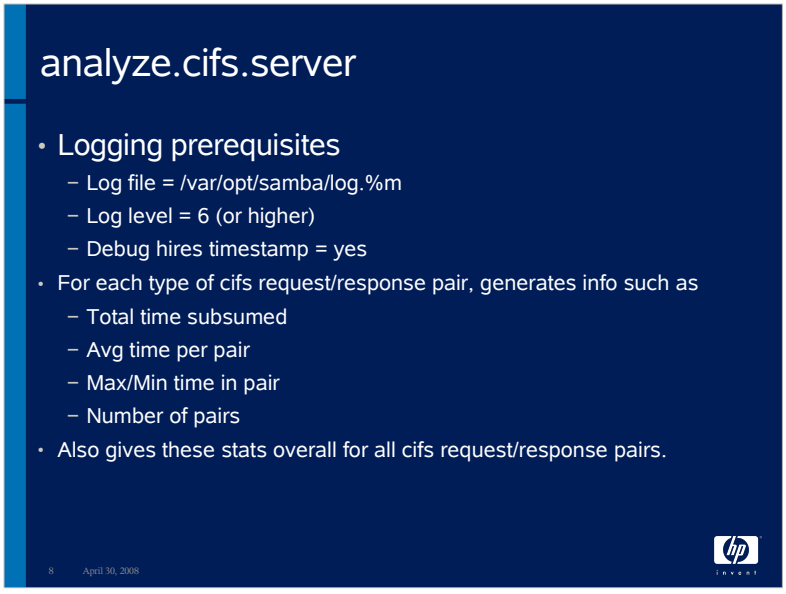
- Sometimes the reason for the delay cannot be determined by the product log file itself. This is often the case when the delay is actually a result of a system call that is not being responded to in an appropriate/timely fashion – with file locking calls, disk writes, and other frequently used system calls, even a small delay may add up to a larger performance problem. For this, the 'spotdelay.tusc' utility can be very useful. This script examines a tusc output file and examines the entry/exit lines in the file to pinpoint system calls that take longer than a specified number of milliseconds. The output of this program identifies individual lines in the tusc output file that bear further examination, as well as the name of the system call identified for the delay.

spotdelay.strace

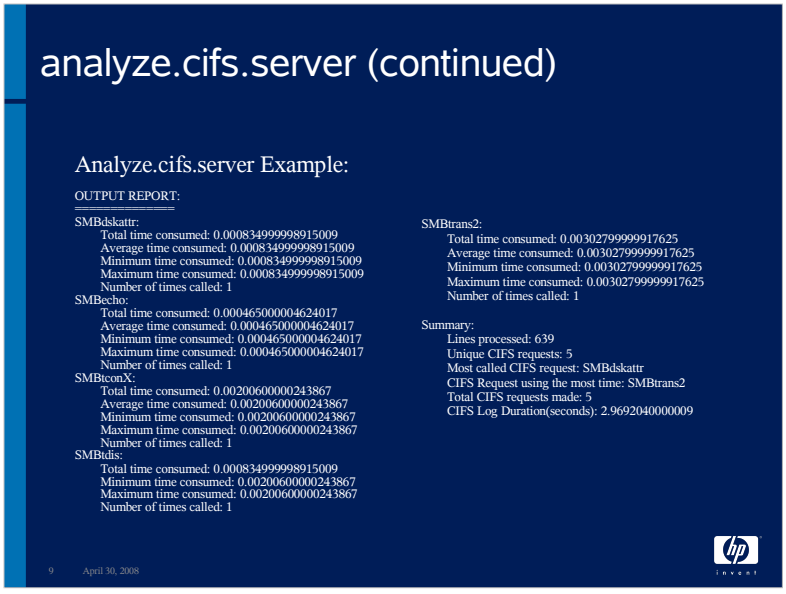
- Logging prerequisites
  - Strace -fttTo <straceoutputfile> -p <smbd pid>
- spotdelay.strace -2 <straceoutputfile
  - scan the straceoutputfile for syscalls with a delay greater than 2 seconds.
- spotdelay.strace –n .002 –s fcntl <straceoutputfile
  - Display lines from strace output with diff of .002 seconds (20 milliseconds) specifically involving the fcntl() system call.

- Because strace has different capabilities than tusc, notably the ability to calculate the time-in-call for each sys call it traps, an implementation for spotdelay.strace is relatively trivial:

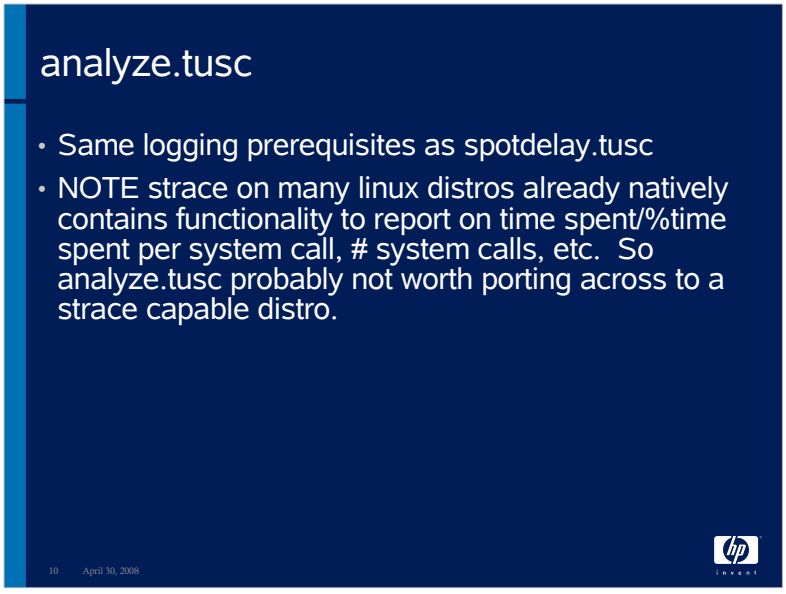
```
#!/usr/bin/perl
# Author: Don McCall, Hewlett-Packard
# Original Creation Date: 4/2008
#
# (c) Copyright Hewlett-Packard 1999
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or (at
# your option) any later version.
#
# This program is distributed in the hope that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See GNU General
# Public License for more details.
#
sub print_help
        {
        print("Usage: spotdelay.strace [-n NN][-s PATT] < straceoutputfile\n");
        print("       where NN is seconds (can be decimal #) delay to look for /n");
        print("       in syscalls where strace was taken with the following options:\n");
        print("        strace -fttTo <straceoutputfile> -p <PID>\n");
        print("       ( delay default is .5 seconds)\n");
        print("       PATT is the system call name you want to look for.\n");
        print("       (PATT default will use all system calls).\n");
        print("       (PATT can be a reg exp,enclosed in double quotes)\n");
        exit(1);
        }
# MAIN
#defaults:
$MAXTIMEDIFF = .5;
$SYSCALLPATT= "[A-Za-z]";
while($_ = $ARGV[0])
        {
        shift;
                if(/^-h/){print_help();}
                if(/^-s/){$SYSCALLPATT=$ARGV[0];}
                if(/^-n/){$MAXTIMEDIFF = $ARGV[0];}
                if(/^-f/){$INPUTFILENAME = $ARGV[0];}
        }
        $totaldelay = 0;
        while ($line=<>) {
        @fishlips = split(' ',$line);
        $time = $fishlips[1];
        $syscall = $fishlips[2];
        @timeincalltemp = split(/\</,$line);
        $timeincall = $timeincalltemp[1];
        ($sec,$fraction) = split(/\./,$time);
        @syscall1=split(/\(/,$syscall);
        @difftemp = split(/\>/,$timeincall);
        $diff=$difftemp[0];
        if($diff > $MAXTIMEDIFF && $syscall1[0] =~ /$SYSCALLPATT/)
        {
        print("line:$.:$syscall1[0]: $diff s\n");
        $totaldelay+=$diff;
        }
                }
        print("Total time spent in syscalls matching criteria  was : $totaldelay\n");
# END
```



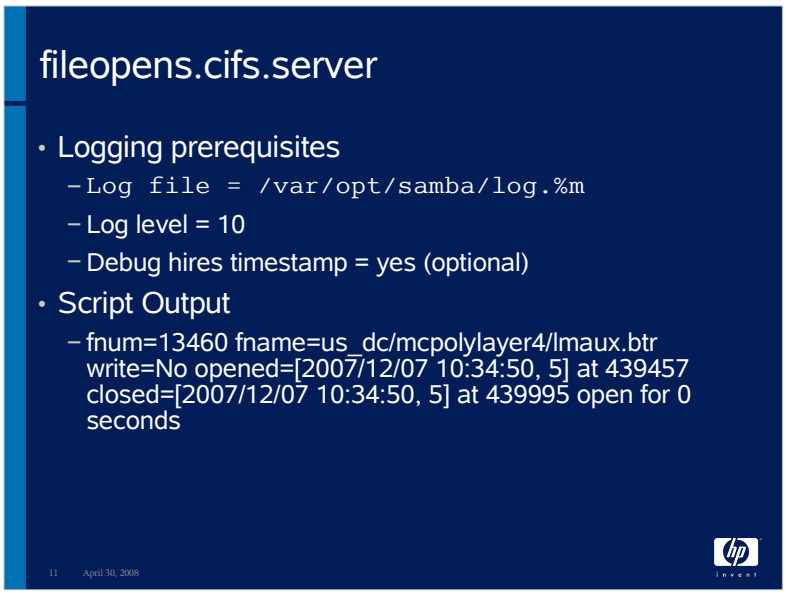
- It is possible that no single CIFS request/response pair is the obvious performance culprit. In this case it is useful to get an overall performance profile of the HP CIFS server process in question. The analyze.cifs.server script keys off of specific well-known debug request/response line items to calculate and report the following information for the time span covered by the debug log file being analyzed:
  - Number of Lines Processed
  - Unique CIFS requests
  - Most called CIFS request (total # of separate requests)
  - CIFS Request using the most time (total time accumulated by one request type across the entire log file)
  - Total CIFS requests made in the entire log file.
  - CIFS Log Duration



- The output on the screen shows a trivial analysis of a log file generated by a connection to a share via smbclient //servername/sharename and a simple dir command on a small directory, and a disconnect from the smbclient.
- This output becomes interesting when you are analyzing a large debug file; Knowing what the most frequently called smb request, and/or the one(s) consuming the most time, can lead to a more in depth investigation of those areas that generate that sort of behavior on the client.



- Analyze.tusc provides a similar function to analyze.cifs.server, but works on a tusc output file rather than a CIFS Server debug log. CIFS requests generally require a number of different system operations to get their job done.
- If analyze.cifs.server pinpoints, for instance, a SMBWrite request as being an issue, running analyze.tusc would help you determine if the delay was a result of the lseek(), write(), fstat() or fcntl() call that may be required to complete the SMBWrite request on the HP-UX system.



Often when you are troubleshooting a performance problem with a customer application that shares files on an HP CIFS Server share it is useful to have a picture of what files are being opened by that application. In addition, knowing how long a file is opened by a particular instance of that application can be useful in tracking down performance delays due to either locking contention or poor file sharing protocols on the part of the application.

Logging prerequisites

Because the script uses a key value (the File Identifier – FID) that is only guaranteed to be unique across a single client session, it is important to configure logging so each client connection to the HP CIFS Server generates it's own log file. You can do this with the `/etc/opt/samba/smb.conf` global parameter:

```
Log file = /var/opt/samba/log.%m
```

Some of the key open and locking statements are only generated in a cifs client session log when the debug logging is set at level 10.

Capture the needed debug statements by setting the `/etc/opt/samba/smb.conf` global parameter:

```
Log level = 10
```

Finally, in database or CAD applications, it is common to find files opened for less that 1 second. If you want to track actual time open for these instances, you must enable micro debugging in the HP CIFS Server product with the `smb.conf` global parameter:

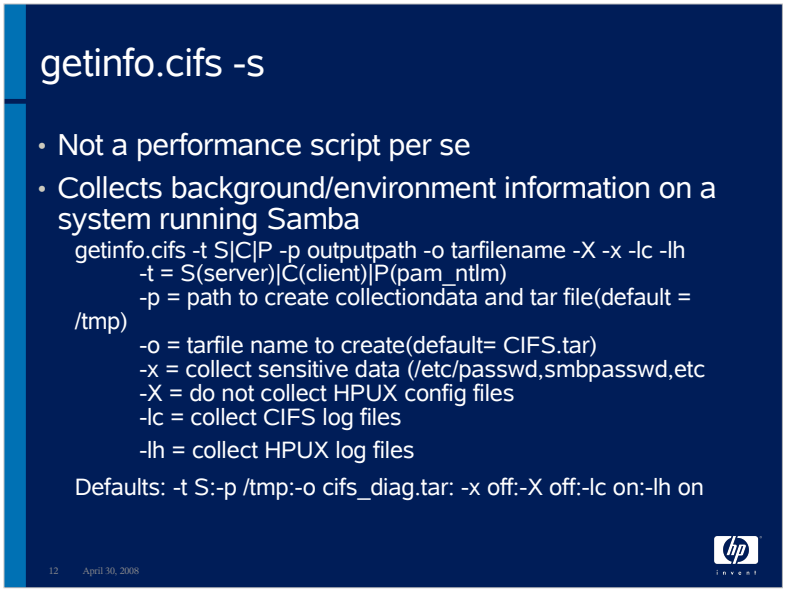
```
Debug hires timestamp = yes
```

Script Output

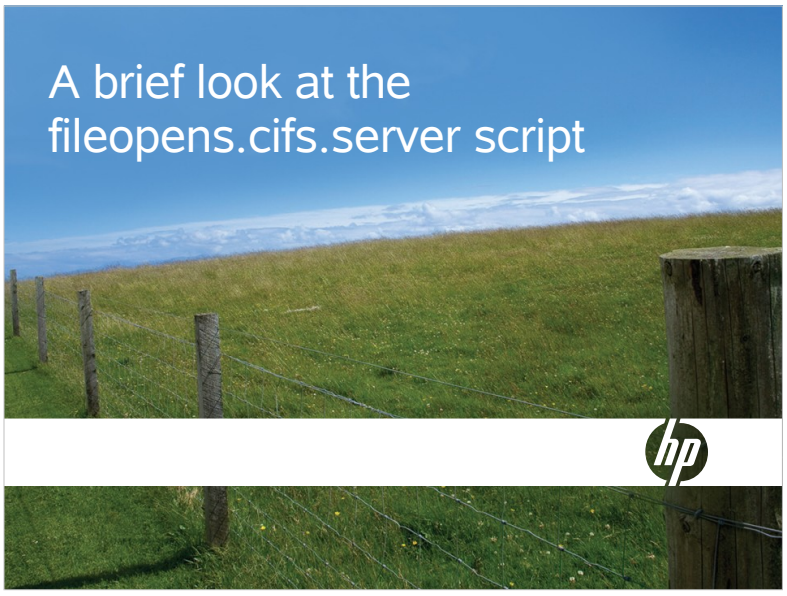
The `openfiles.cifs.server` script will output lines similar to those in Figure 1:

fnum=13460 fname=us_dc/mcpolylayer4/lmaux.btr write=No opened=[2007/12/07 10:34:50, 5] at 439457 closed=[2007/12/07 10:34:50, 5] at 439995 open for 0 seconds

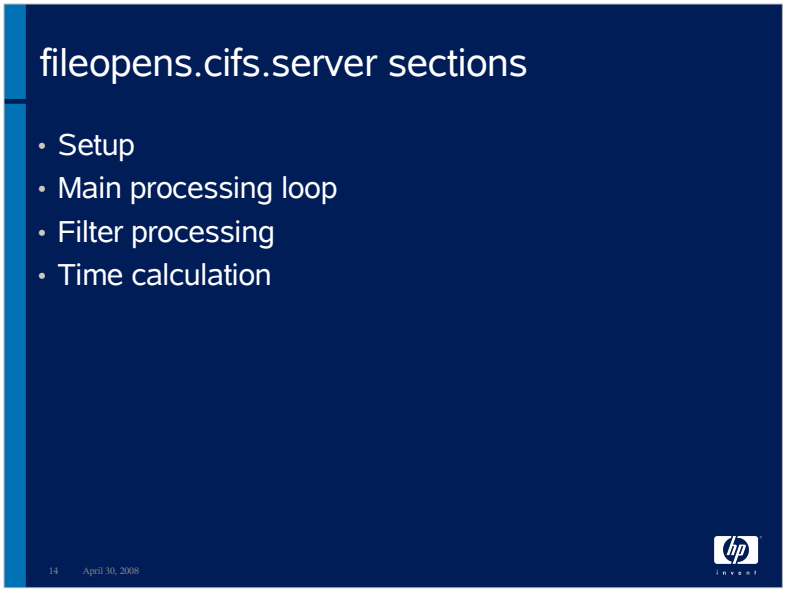
Fnum is the unique file number assigned by the HP CIFS Server smbd daemon when the file is open, and is unique for this client session.
Fname is the file path and name relative to the root share path of the share it resides on.
Write = [Yes/No/DIR] indicates whether the entry is a directory (DIR) or if the file was opened for write access.
Opened is the timestamp reported when the file was opened.
At ###### is the line number the open or close occurred on (useful for examining a particular open/close sequence in detail in the log file).
Closed is the timestamp for the close of the file.
Open for is the number of seconds the file remained open.
You may also see two other possible output permutations. The first, shown in Figure 2 below:
fnum=13387 fname=us_dc/mcpolylayer1/lmlandmark.rec write=No opened=[2007/12/07 10:34:49, 5] at 382054 closed= at open for -38089 seconds
Figure 2: output line for an unclosed file
This indicates a file that was STILL open at the time of the termination of the debug log file. Note the null field for 'closed=' and the negative number displayed in 'open for'.
The other permutation you may see depends on 'debug hires timestamp = yes' and is shown in Figure 3:
fnum=13094 fname=us_dc/mcshapes.rec write=No opened=[2008/01/09 09:09:46.615247, 5] at 739462 closed=[2008/01/09 09:09:46.616863, 5] at 739572 open for 0.00161600000137696 seconds
Figure 3: output line with debug hires timestamp enabled
Note the fractional second output in the 'open for' field.



- getinfo.cifs -s doesn't strictly speaking belong in a performance problem toolset, but it is very useful as an initial step to get some understanding of the environment that the HP CIFS Server resides in. The files and information it collects from the system can be very useful in understanding what external pressures may be involved in the Samba performance footprint. Depending on what commandline options you give it, it can collect such things as:
    - /etc/nsswitch.conf
    - /etc/krb5.conf
    - Ls -l outputs of important samba directories like the (locks dir, for instance)
    - Uname -a
    - Rpcinfo -p
    - Smbd -V
    - Etc...
- It can also collect syslog and samba debug files, as well as more sensitive system files (such as /etc/passwd, /etc/group, etc). But these collections must be explicitly requested, as they can lead to very LARGE collections, as well as putting sensitive data into a tar file where it could be used inappropriately if not safeguarded properly.



- In this next section of the talk I would like to briefly step through the code of one of these scripts, to give you an idea of how they are written, and perhaps help you get started with generating or modifying scripts for your particular Samba platform or problem.
- Confession time; as I started off saying, these scripts 'evolved' as one-off tools used in a particular problem senario. As such, there was no requirement for consistency in writing style, etc between the scripts.
- The fileopens.cifs.server script is one of the later scripts, so I am going to use that as a basis for our discussion here, as it was written with a little bit more care to attempt to generate more 'reusable' code.



•The `fileopens.cifs.server` Perl script consists of four parts: a setup section, a main processing loop, and two subroutines. The script was written in such a way that it should be possible to adapt it to work on other timestamped application log files, as long as the user has a good understanding of the format of these files, and there is a unique 'key' that can be used to group together debug lines related to the operation to be studied. Note that this version of the script will have some options (like output in CSV format) that are not described here for the sake of simplicity.

•The variables in the setup section are used to hold the various 'filters' that will be applied to a log entry. This determines what sort of information it holds.

• For our fileopens script, we use five such filters:

•timestamp lines (timeentry_grepper),

•the key we will use to group our information (key_grepper),

•the file open (starttime_grepper),

• the file close (endtime_grepper) and

• the write access type for the file open (auxinfo_grepper).

• Setting appropriate values for these variables requires an in-depth familiarity with the application log files, and must only return the lines that you expect. Using the 'grep' command on the log file with the proposed filters was critical in determining the appropriate phrases to use to isolate the debug log file entries we needed.

Filter processing

- The `process_filter` routine takes the following parameters:
  - a search string
  - value associative array
  - time associative array
  - line number associative array
  - index value or offset (depends on last passed parameter)
  - current line position (base 0) for associative array value
  - "value" or "index" determines whether index value or index position is passed into the routine.

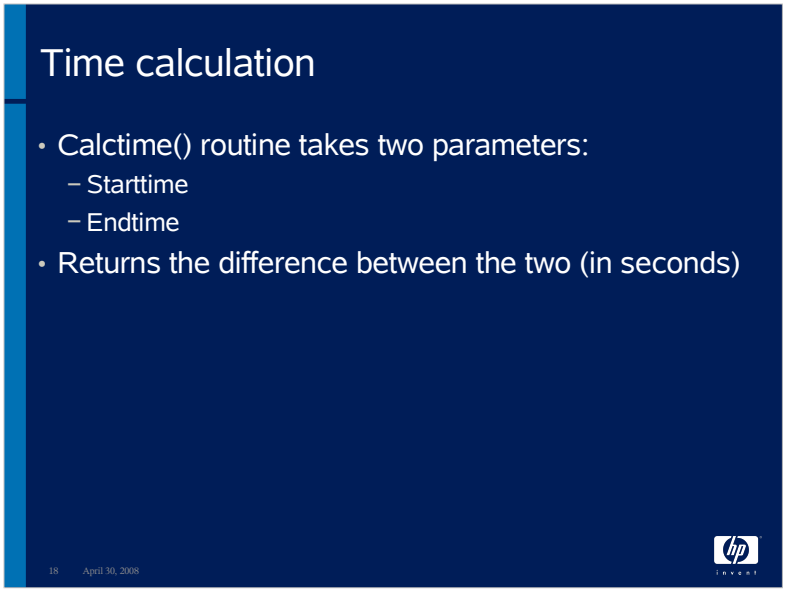
16 April 30, 2008

•What the routine does is associate the key (the fnum) with the timestamp, the value (for instance for the readwrites value array, the value is either ‘Yes’, ‘No’, or ‘DIR’), and the line number of the debug entry where the event occurred, based on either a passed value, or an index into the current line, depending on whether parameter 6 is ‘V’ or ‘I’ respectively. It uses the first parameter (search string) to determine whether this line should be processed or not.

•NOTE: it would probably be more efficient to move the search string OUT of the process_filter() routine, and test inline in the Main Loop. But I have left the code this way for purposes of clarity and adaptability.

Filter processing

```
sub process_filter {
        $line_to_process = "";
        if( /$_[0]/ ){
        print "process_filter: _[6] = -$_[6]-\n" if $debug;
        @line_to_process = split;
        if($_[6] =~ "I"){
        print "process_filter: its an index!\n" if $debug;
        $pindex = $line_to_process[$_[4]];
        } else {
        $pindex = $_[4];
        }
        $_[1]{$pindex} = $line_to_process[$_[5]];
        $_[2]{$pindex} = $timeline[0] . "I" ;
        $_[3]{$pindex} = $.;
        print "process_filter: key=$pindex, a1{key}=$_[1]{$pindex},a2{key}=$_[2]{$pindex},a3{key}=$_[3]{$pindex}\n" if $debug;
                }
}
```

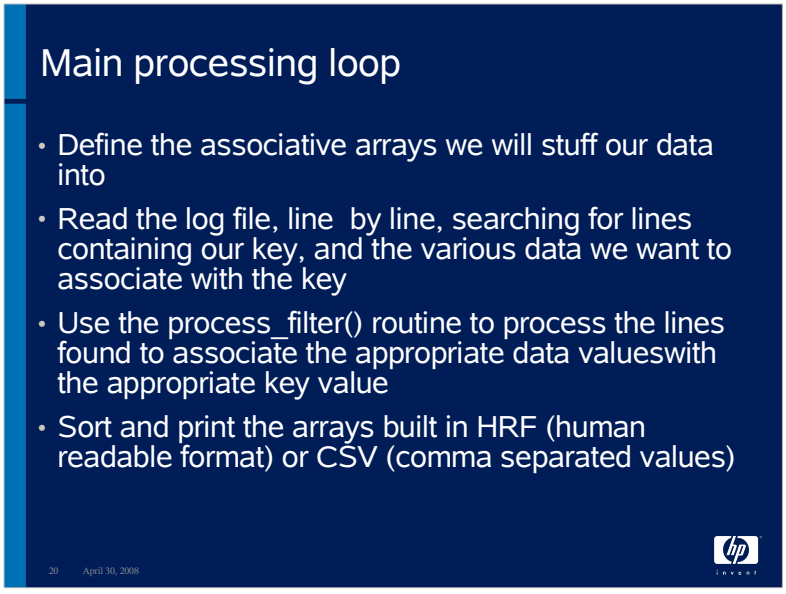


•The `calctime` routine simply strips the time information from the date/time stamp entries it is passed, and returns a difference between the two in seconds. It expects the higher (e.g. later) timestamp as the second parameter. This routine is split out primarily for ease of modification in the event that the timestamp format for the log file changes in later revisions, or in the event this script is adapted to other application log files that use a different timestamp format.

Time Calculation

```
sub calctime {

        ($opentime,$closetime) = @_;
        local($discard, $opened, $discard2) = split / /,$opentime;
        local($discard, $closed, $discard2) = split / /,$closetime;
                local($hour, $minutes, $seconds) = split ":", $opened;
                local($calcopentime) = ($hour*3600) + ($minutes*60) + $seconds;
                local($hour, $minutes, $seconds) = split ":", $closed;
                $calcclosetime = ($hour*3600)+($minutes*60)+$seconds;
                print "calctime: open time: $calcopentime\n" if $debug;
                print "calctime: close time: $calcclosetime\n" if $debug;
                $diff = $calcclosetime - $calcopentime;
                print "calctime: Diff=$diff\n" if $debug;
        return $diff;
}
```



•Here we read lines in from the debug file, and using the various filters we defined in the setup section, we search for and process the lines that contain the key value (fnum, in our case) and the various data points we want to capture about each file open. Note that the majority of the 'processing' of these lines is done in the 'process_filter' routine. In the main loop, we define the associative arrays and identify the lines that are then fed to the process_filter routine to build the entries in the various associative arrays we use at the end of the Main Loop to print out the open file information.

Main Loop

```
# MAIN loop:
while (<>) {
        chomp;
        if ( /$timeentry_grepper/ ){
                @timeline = split(/\]/,$_);
                next;
                } else {
# get fnum for this file
        if ( /$key_grepper/ ){
        @temp = split;
        $current_fnum=$temp[6];
        print "current_fnum = $current_fnum\n" if $debug;
        }
#get the open info for this file:
        process_filter($starttime_grepper,*opens,*opened_time,*opened_line,$current_fnum,7,V);
# get the read and write access.  NOTE if a directory, this will not be
# available.  This will result in a readwrites{} of null for the current_fnum.
        process_filter($auxinfo_grepper,*readwrites,*readwrites_time,*readwrites_line,$current_fnum,5,V);
# now process the close and get the closed time and closed line:
        process_filter($endtime_grepper,*closes,*closed_time,*closed_line,3,4,I);
        }
}
#Print out the results in HRF (human readable format)
        foreach $key (sort keys(%opens)) {
        print "fnum=$key fname=$opens{$key} ";
        if($readwrites{$key}) {
                print $readwrites{$key};
                }else{
                print "DIR";
                }
        print " opened=$opened_time{$key} at $opened_line{$key}";
        print " closed=$closed_time{$key} at $closed_line{$key}";
        print " open for " . calctime($opened_time{$key},$closed_time{$key}) . " seconds\n";
        }
```

Where to get the scripts?

- If you have an A.02.02 or greater installation of HP CIFS Server (Samba), the tools and man pages can be had from the following directory:
  - opt/samba/WTEC_Support_Tools
- Download the HP CIFS Server product from
  - http://software.hp.com
    - 'Internet ready and networking
      - HP CIFS Server
  - This will give you a .depot file, which is basically just a tar file; extract the WTEC_Support_Tools directory via
    - Tar -xvf HPUXxxxxxxxx.depot CIFS-Server/CIFS-UTIL/opt/samba/WTEC_Support_Tools

Thanks for your Time!