



HP-UX 11i Knowledge-on-Demand

HP technical Webcast series: software optimization



Technology for better business outcomes

HP-UX 11i v3 Knowledge-on-Demand

- Objective: Support software development partners and customers in achieving better business outcomes with HP-UX 11i.
- What HP is providing: a series of technical on-demand training Webcasts
 - Focused on helping developers increase performance through application optimization for HP-UX 11i v3 on HP Integrity servers
 - Access to HP for follow-up questions
 - Available at www.hp.com/go/knowledgeondemand

HP-UX 11i v3 Knowledge-on-Demand Webinars – planned curriculum

- Foundation Track
 - Module 1: How to upgrade to HP-UX 11i v3
 - Module 2: HP-UX open source resources
 - Module 3: Unified file cache
 - Module 4: Caliper
 - Module 5: NUMA Tuning: Getting the Most Out of Your Cellular Server by using NUMA
 - Module 6: The Mercury Library – Increasing Application Performance
 - Module 7: Software Transition Kit's (STK's) for HP-UX 11i v3
- Java Developers Track
 - Module 8: Java Memory Management - Internals and Performance
 - Module 9: HPjmeter – measure Java application performance on HP-UX 11i
 - Module 10: Solving Java performance problems
- C/C++ Developers Track
 - Module 11: pthreads enhancements in HP-UX 11i v3
 - Module 12: Kernel tracing & profiling tools (internal tools)
 - Module 13: Using compilers to get optimal performance
 - Module 14: HP Code Advisor: A Powerful New C/C++ Analysis Tool for HP-UX
 - Module 15: Montecito Hyper-Threading on HP-UX 11i v3

Additional Webinars
published going forward!

Related HP-UX 11i v3 resources

- All developers' resources
 - HP-UX 11i developers' content
www.hp.com/go/hpuxdev
 - HP-UX 11i v3 news, functionality, product download and services resources
www.hp.com/go/hpux11i
 - HP Integrity server ISV resources for DSPP members
www.hp.com/go/dspp_integrity
 - HP Integrity server product information
www.hp.com/go/integrity
- Software partner promotional opportunity
 - HP promotion for HP-UX 11i v3-ready software partner application
www.hp.com/go/v3promotion

Enjoy this Knowledge-on-Demand topic!

Thank you for taking time to learn about HP-UX 11i v3 and related technologies.

Please send comments on today's topic and/or requests for future topics to:

hpuxquestions@hp.com



The Mercury Library: Increasing application performance

An HP-UX 11i Knowledge-on-Demand software optimization Webcast



Technology for better business outcomes

Introducing today's speaker

Douglas Larson has been a technical leader in the HP-UX kernel lab and benchmarking organization for over 15 years. Previously to that he worked in the HP-UX kernel lab on process management, virtual memory management, and other areas for almost 10 years. He is also considered one of the lab's leading experts on multiprocessor software performance, locking, and locking contention in particular.

He developed some of the most widely used internal HP-UX performance tools (e.g. spinwatcher, ktracer, kernel profiling). He works with internal HP organizations as well as field organizations and customers to improve HP-UX and customer performance.

Doug holds two computer-performance related patents.



Agenda

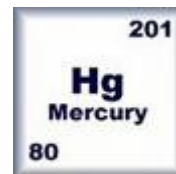
- Mercury rationale
- Mercury overview
- Mercury private functions
- Mercury public thread information
- Mercury caveats
- How fast is Mercury?
- Summary

Introduction: Mercury rationale

- Customer and field reports of certain heavily used areas of functionality
 - Tracking time
 - User space locking contention issues
 - Both too expensive in terms of CPU time
- System call overhead is one key problem here.
 - Calls are done many, many times.
 - Calls are done in performance-critical parts of the code.
- **Solution:** Let almost all the work be done in user space.
- Another key problem is user space lock contention when many threads are sharing locks and resources.
 - Threads waiting for locks spin when the lock holder is not running—wasteful.
 - Threads holding locks/resources are context-switched out by the kernel, causing waiting threads to have to wait even more—once again, wasteful.
- **Solution:**
 - Let threads see the run-state of other threads.
 - Let threads in critical sections prevent themselves from being switched out.

The Mercury name

- “Mercury” was chosen as a name independent of any other use within HP.
 - Mercury was the god of speed in Roman mythology. In particular, he was the messenger of the gods.
 - Mercury is an element (also called “quicksilver”), given the chemical symbol “Hg.” All Mercury function names begin with the prefix “hg_”.



Mercury overview: general

- Mercury consists of an HP-UX kernel part and a library part.
 - The kernel part is fully implemented in HP-UX 11i v3 (11.31).
 - The library part is also implemented in the same release. The library can be compiled into a C program by using the “-lhg” option.
 - The interface consists of a set of about 16 function calls that are accessible from user programs.
 - Mercury has a manual page—just type in “man mercury” on your 11i v3 system.

Mercury overview: private functions

- Mercury thread-private functions
 - Have a per-thread memory area that the kernel knows about; used by the Mercury library, not directly by user code
 - If changed, the kernel updates when returning to user space after a system call or after a trap has occurred.
 - Consist of various time-related functions, various context switch related functions (including setting critical regions), and getting the CPU number you are running on
 - Run fast because they are mostly just memory operations, not system calls

Mercury overview: public functions

Mercury public functions

- Have a memory region shared with all threads in the system. This area is accessed by the Mercury library only, never directly by the user code. This region is read-only for user threads.
- The kernel updates the information in this region as it happens, on a “best effort” basis.
- Only threads that request it will be tracked in this global memory region.
- Allow one thread to see what state other threads are in as well as allow a thread to become a thread with a viewable state
- Run fast because they are mostly memory operations, not system calls
- Have as their original purpose promoting the efficient use of shared resources

Mercury private: time functions

- `hg_gethrcycles()`: time in machine-dependent cycles since the machine has booted; adjusted to match Monarch
- `hg_gethrtime()`: time in nanoseconds since the machine booted; adjusted to match Monarch; very comparable to the widely used `gethrtime()`, only faster
- `hg_nano_to_cycle_ratio()`: useful if you wish to record your raw data in cycles (because `hg_gethrcycles` is the fastest) and later convert them to nanoseconds for output
- `hg_busywait()`: just spin wait for a fixed amount of real time; changing processors will not throw this off; a problem solved compared to spin waiting using the ITC

Mercury private: context switch functions

- `hg_context_switch_involuntary()/hg_context_switch_voluntary`: returns number of involuntary/voluntary context switches since the thread creation
- `hg_context_switch_tries()`: involuntary + voluntary
- `hg_setcrit()`: requests the kernel to start (or stop) taking involuntary context switches; more on `hg_setcrit()` next page
- A context switch, in the HP-UX kernel, occurs when a processor begins to execute a thread (“switch in”) or when it ceases to execute a thread (“switch out”).
 - A voluntary context switch occurs when a thread switches out as a result of actions it is taking—e.g., reading from a disk, or taking a page fault.
 - An involuntary context switch occurs when a thread is switched out as a result of kernel policy decisions—e.g., end of a time slice, or a higher-priority thread has become runnable.

Mercury private: hg_setcrit() details

hg_setcrit() can act to prevent the thread from taking involuntary context switches.

- Useful for reducing contention on locks of critical regions
 - This helps solve the problem that if a thread holds a lock while it is switched out and the lock hold time goes up, causing other threads to have to wait for it. It is better for the lock holder to finish as fast as possible.
- The kernel will usually honor this request, but policy reasons may require the kernel to do a context switch anyway. Examples:
 - The thread may have held off too many switches; for fairness reasons, the kernel will begin to override the requests.
 - A higher-priority real-time thread may have become runnable. Higher priority time-share threads won't do this.
- And others ...
- I recommend using the following construction (different from man page):

```
hg_setcrit( lock_address, UNWILLING_TO_BLOCK)
acquire lock
critical region
release lock
hg_setcrit( CRIT_OFF, UNWILLING_TO_BLOCK )
```

If your critical regions are short enough compared to your other code, this will be a win for you relative to the sequences on the man page; you will take enough context switches elsewhere to keep the kernel happy, and you won't have to take extra ones here.

Mercury public: structure

- There is a memory region shared with all threads in the system. This area is accessed by the Mercury library only, never directly by the user code.
- The kernel updates the information in this region as it happens, on a “best effort” basis.
- Only threads that request it will be tracked in this global memory region.
- Some Mercury public functions see what state other threads are in; others control when a thread has its state viewable.
- The functions run fast because they are mostly memory operations, not system calls.

Mercury public: initialization

- `hg_public_init()`, `hg_public_remove()`: Allows a thread to make its state viewable (and get a viewing handle), or turn off viewability
 - Initializing is only needed if the calling thread needs to have its own state viewed by other threads. Viewing the state of another thread only requires knowing its handle.
 - The handle returned by `hg_public_init()` must be passed to the threads who wish to view the state. This is the only source of the handle. One way to pass this handle is to put it in a shared lock structure so that the thread waiting can view the handle owner.
 - Remember, once a thread has made its state viewable, any thread on the system may view it. Not knowing the handle is only a minor barrier for someone determined to view that state.

Mercury public: viewing state

- `Hg_public_is_reporting()`: returns true if the given handle is associated with an existing thread. If a thread has exited, it is automatically removed from the reporting state.
- `Hg_public_is_running()`: returns true if the associated thread is currently executing on a processor
- `Hg_public_is_onRunQ()`: returns true if the associated thread is currently runnable, but waiting on a run queue
- NOTE: Handles are re-used! Therefore care must be taken to ensure that the thread which you expected to be associated with the handle, hasn't died and had its place taken by a different thread.

How fast is Mercury?

- **Time calls:** As an example, let us consider calls to get time, the standard `gettimeofday()`, the less standard but widely used `gethrtime()`, and the HP Mercury call `hg_gethrtime()`:
 - `gethrtime()` is about 5 times faster than `gettimeofday()`.
 - `hg_gethrtime()` is about 3 times faster than `gethrtime()`.
 - `hg_gethrcycles()` is about 25% faster than `hg_gethrtime()`.

How fast is Mercury?

- **State and context switch calls:** Colin Honess, a pre-sales engineer in the war room, wrote user space code to replace parts of the `fcntl()` system call for a customer. Basically, the replacement was to write locking code in user space. The speedups he got for a test case that emulated a lock-intensive part of the user code was as follows (speedup versus `fcntl()` locking):
 - Uncontended lock: 4x
 - 4 processes: 20x
 - 32 processes: 35x
 - 64 processes: 61x
- Although Mercury enabled this result by making certain operations possible in user space, obviously most of the credit must go to Colin's clever code.

Mercury caveats

- The Mercury interface is HP proprietary.
- Mercury values returned are to be considered hints.
 - For example, values returned by `hg_gethrcycles()`, which are adjusted to return comparable value on all processors, may be off by a few cycles because the processors' interval timers may drift with respect to each other.
- Mercury advisories are to be considered hints to the system.
 - For example, when calling `hg_setcrit()`, normally the kernel will honor the request and not context switch the thread out. However, the kernel at some point may decide not to honor this request. One reason is that if context switching has been held off too much by this thread, it will no longer be honored.

Summary

- Use Mercury time functions to speed up applications requiring rapid, frequent use of getting time.
- Use Mercury context switching control and thread state viewing functions to help reduce contention for user-written locks.
- Don't use Mercury for timestamps generated by multiple threads that must be in the right order.
- Don't use Mercury context switch control for code that requires such control for functional correctness; it is a hint to the OS, not a command.