HP OpenVMS technical journal V19

Multi-threading: concepts and application tuning

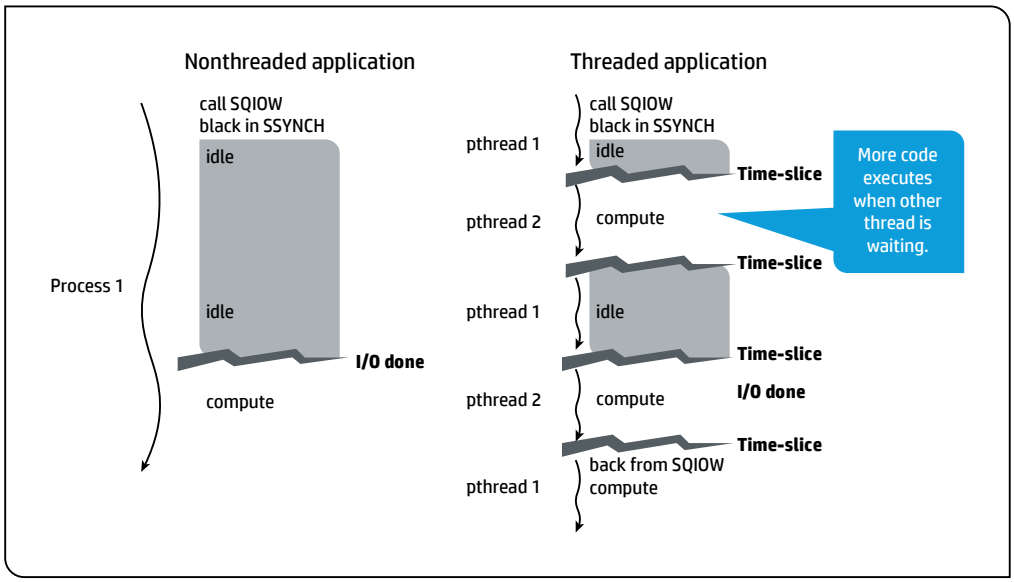
Table of contents

With the increasing number of cores in systems, users can improve the performance of their applications by effectively utilizing this additional computing power. Effective utilization of multicore systems involves writing multi-threaded applications. This paper discusses basic concepts and principles necessary to write efficient multi-threaded applications. It also discusses tuning threaded applications to get maximum performance.

Introduction to multi-threading

Multi-threaded programming involves organizing and coding a program so that instances of its routines, called threads, can execute concurrently in the same process. Threading can be used to improve a program's performance—its throughput, computational speed, responsiveness, or a combination of any of these. Using threads can improve a program's performance on uniprocessor systems by permitting the overlap of input, output, or other slow operations with computational operations. A multi-threaded program can perform other useful work while waiting for slower operations like I/O completion to produce its next event. A program with multiple threads can be especially suited to run on a multiprocessor system, where threads run concurrently on separate processors.

Figure 1. Nonthreaded vs. threaded application execution

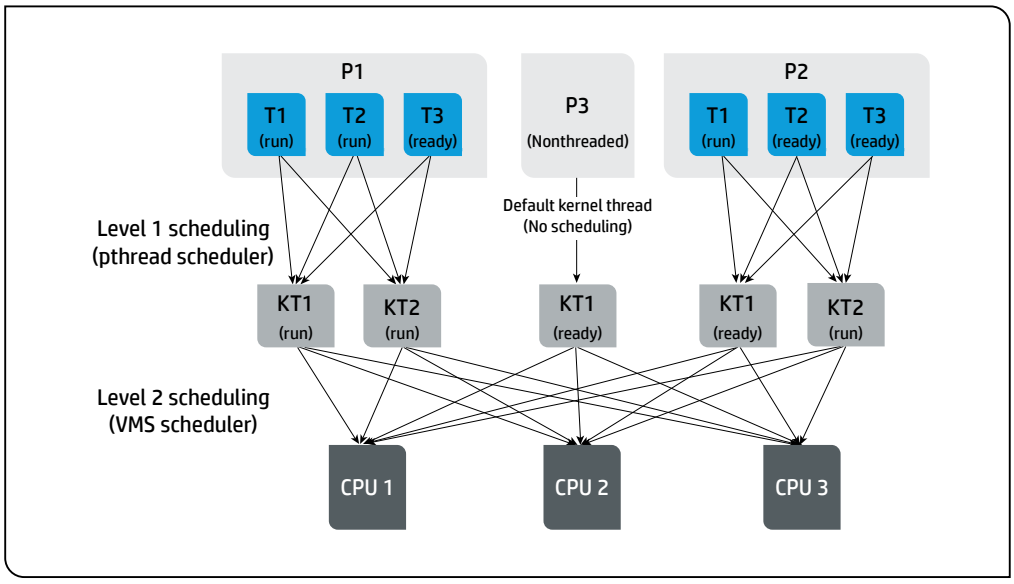


User and kernel threads

A thread is a single, sequential flow of control within a process. Within each thread there is a single point of execution. An application on HP OpenVMS can create and manage threads by using the pthreads library available with the operating system. These threads that are created by the application are called user-threads or pthreads. Figure 1 above shows the difference in execution between a nonthreaded and a threaded application.

Kernel thread is a logical entity on which user threads are scheduled by the threads library, and in turn, kernel threads are scheduled on the processors by the OpenVMS scheduler. Kernel threads were introduced in the OpenVMS Executive to allow threads rather than processes to be scheduling entities and hence effectively utilize multiple processors. Each process can have multiple kernel threads, but this number is limited to the number of processors on the system. OpenVMS scheduler now schedules the kernel thread (instead of a process), and hence, multiple user threads can execute simultaneously. This is illustrated in figure 2. Processes P1 and P2 are threaded, contained three user threads each, T1, T2, and T3. Both the processes have two kernel threads, KT1 and KT2. The corresponding scheduling states are mentioned within brackets. P3 is a nonthreaded process having the default kernel thread KT1. There are three CPUs: CPU1, CPU2, and CPU3. The black arrows illustrate the various scheduling possibilities, and the red arrows illustrate a particular time instant when T1 and T2 of process P1 and T1 of process P2 are running.

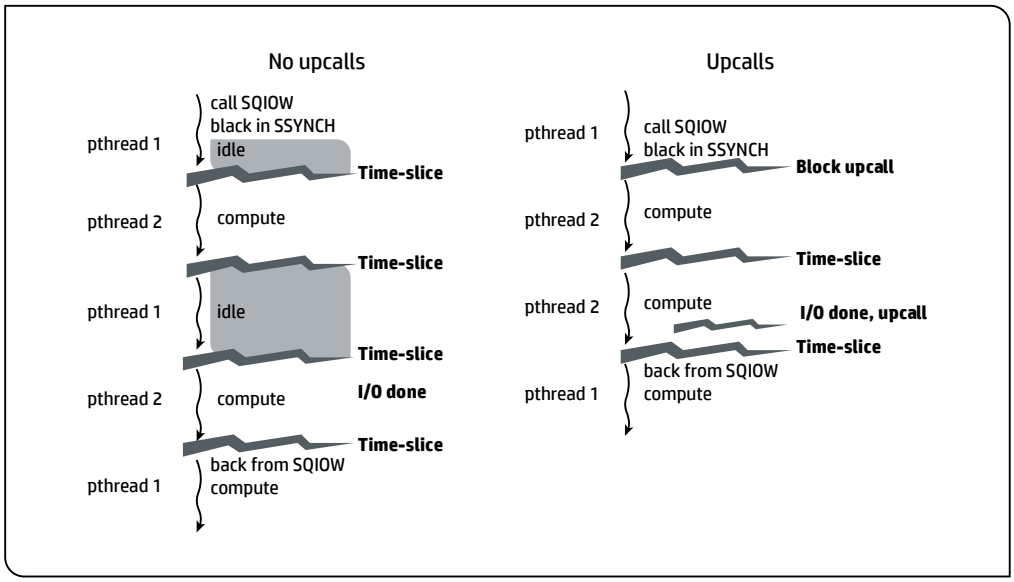
Figure 2. Kernel threads



Upcalls

We have now seen how the usage of pthreads could make better utilization of the processor cycles. However, if you observe figure 1 carefully, you can find that there are still some scenarios where the processor cycles are wasted. For example, let us assume that thread 1 has issued an I/O and is waiting for its completion. While it is waiting, thread 2 gets scheduled. However, threads library does not know if the I/O is completed or not. Hence during the next time-slice, thread 1 is selected again for execution. But, thread 1 is still waiting for the I/O to complete. Hence it will waste the processor cycles when it is scheduled. To further improve the CPU utilization, the concept of "upcalls" was introduced. When upcalls are enabled, if a thread issues an I/O, instead of waiting and wasting the processor cycle, the kernel issues an upcall to the threads library. Upcall is a mechanism where the kernel communicates an event back to the threads library. In this example, the kernel issues an upcall to the library saying that the thread is waiting for an I/O. The threads library recognizes this, and takes the thread out of execution, and schedules some other thread. In this way, the processor is utilized much more effectively. Figure 3 shows the difference between the application behavior when upcalls are enabled and disabled.

Figure 3. No upcalls vs. upcalls



Application performance

Multi-threaded applications can run in several configurations:

- No upcalls
- Upcalls with one kernel thread
- Upcalls with multiple kernel threads

The performance of the application depends on which of these modes the application is running in as well as the type of application. For example, we have already seen the benefits of enabling upcalls. Let us assume that the application has only one pthread. With such an application, enabling upcalls might slow down the performance of the application. With no upcalls, the thread would just block. However, with upcalls enabled, the system services would do some extra communication back to the threads library, and the library has to select some other thread for execution. Similarly, after the I/O is complete, kernel sends an upcall back to the threads library and it has to schedule the waiting thread back. If the thread is I/O bound, then the performance would be worse. Another example where enabling upcalls would degrade the performance is when you have two threads communicating to each other using blocking calls. Typical examples of this would include thread synchronization using mutexes or locks. Such scenarios involve lot of extra kernel code to be executed to signal upcalls, which could impact the performance.

In a similar way, the performance of the application varies depending on whether multiple kernel threads are enabled or not. If the application has a lot of user threads, and each can be run in parallel, then the application performs better by having multiple kernel threads. However, it is not always true that applications run faster with higher number of kernel threads. If there are multiple user threads, and each of these wait for the same mutex to execute, then the performance could degrade with multiple kernel threads. Hence, depending on the type of application, the number of kernel threads should be limited. For example, few applications perform better with four to five kernel threads, but the performance of some applications start degrading when the number of kernel threads is increased beyond two.

To get maximum performance benefit, the application has to be designed in such a way that there are things that can run in parallel. If most of the threads in the application rely heavily on serializing the events (by holding a mutex or condition variable), then it is recommended that the application be run with just one or two kernel threads. However, if there are multiple threads that can run independently, these applications would give a better performance when more kernel threads are enabled. To summarize, application performance varies depending on multiple factors as discussed above. It is ideal to check your application with various combinations of upcalls/number of kernel threads to figure out what works best for the specific application.

Application tuning

On OpenVMS, the image header of an executable image has two flags to tailor the behavior of threading.

- Use upcalls
- Use multiple kernel threads

As these flags are part of the image header, these flags could be controlled during link time. You can use the following qualifier to control the threading behavior.

$ LINK/THREADS_ENABLE=(MULTIPLE_KERNEL_THREADS, UPCALLS) ABC.EXE

In this example, ABC.EXE is linked with the flags UPCALLS and MULTIPLE_KERNEL_THREADS. When this image runs, it would have upcalls enabled, and can have multiple kernel threads. The following table describes the various combinations allowed.

Upcalls	Multiple kernel threads	Remarks
Disabled	Disabled	Allowed
Disabled	Enabled	Illegal: MKT flag is ignored
Enabled	Disabled	Allowed
Enabled	Enabled	Allowed

The threading behavior can be altered dynamically once the image is created by using the following utility.

- On HP Alpha system

  The THREADCP utility can be used to alter the image header flags that control the threading behavior.
  $ THREADCP ABC.EXE /ENABLE = (UPCALL, MULTIPLE_KERNEL_THREADS)

  You can also view the behavior using this utility.
  $ THREADCP ABC.EXE /SHOW
  SYS$SYSDEVICE:[USER_DIRECTORY]ABC.EXE;1
  %THREADCP-I-MKT, multiple kernel threads are enabled
  %THREADCP-I-UPC, upcalls are enabled

- On HP Integrity system

  SET/SHOW IMAGE commands can be used as the THREADCP utility is not available on Integrity platforms. For example,
  $ SET IMAGE ABC.EXE /FLAGS = (UPCALLS, MKTHREADS)
  $ SHOW IMAGE ABC.EXE
  . . .
  . . .
  Image Link Flag Description
  --- ---
  MKTHREADS : Multiple kernel threads enabled
  UPCALLS : Upcalls enabled
  . . .

In addition to the image header flags, there is a SYSGEN parameter called MULTITHREAD that overrides these image header flags. The behavior of the applications depends on the value of MULTITHREAD parameter (This parameter is dynamic from HP OpenVMS V8.3).

Multi-thread	**Description**
0	No upcalls
1	Upcalls, single kernel thread
>1	Number of kernel threads

By default, the value of MULTITHREAD parameter is set to the number of active processors on the system.

Note that the image header flags only specify whether the application can have multiple kernel threads or not. The number of kernel threads is limited by MULTITHREAD parameter. One of the limitations of this approach is that all the threaded applications in the system can have a maximum of “MULTITHREAD” kernel threads. If there are multiple threaded applications on the system, and each requires a different number of kernel threads to run efficiently, it was not possible to have different kernel thread limit to each process/application. However, with OpenVMS V8.4, a new feature was added to control the number of kernel threads on a per-process basis. The upper limit, however, is controlled by MULTITHREAD. The following user interfaces could be used to make use of this feature:

- SYS$CREPRC
- SYS$SET_PROCESS_PROPERTIESW
- SET PROCESS/KERNEL_THREAD_LIMIT=n
- RUN/KERNEL_THREAD_LIMIT=n
- SPAWN/KERNEL_THREAD_LIMIT=n
- LIB$SPAWN

Other considerations in a threaded environment

ASTs

Traditionally, OpenVMS supports Asynchronous System Traps (ASTs) for concurrent execution. The threads programming model also allows asynchronous execution without the usage of ASTs. For example, instead of issuing a $QIO and then doing the completion in an AST, the program could create and join with a thread. Nonetheless, ASTs may still be used in a threaded environment. However, note that under certain conditions (for example on a multiprocessor environment) it is possible to have one AST executing concurrently with one or more normal level threads. Hence on a multi-kernel-threaded environment, being at an AST level doesn't mean that the AST locks out the normal code execution.

When a user-mode AST is ready to be delivered, it generally uses the user stack of the process to execute the AST. In a threaded environment, when upcalls are disabled, the threads library is totally unaware of the AST activity in the process. Hence the AST executes on the stack of the thread that happens to be running at the time of AST delivery. Because the created thread stacks are relatively small (by default), there is a risk of the thread running out of its stack when an unexpected AST consumes even moderate amount of its stack. When upcalls are enabled, the kernel notifies the threads library that an AST is ready to be delivered. When the threads library receives such an upcall, it forces the default thread to be scheduled in, and delivers the AST on the default thread stack. As the default thread has huge (theoretically almost 1 GB) stack space, there is minimal risk of overflowing the thread stack.

Each pthread maintains its own AST enable state that is part of the context that is saved on every thread context switch. When upcalls are disabled, and the threads library is unaware of AST activity, this could lead to issues. For example, if the thread had disabled ASTs, and later blocked for some reason, it would be taken out of execution. Now another thread can have ASTs enabled and can execute. The application should be prepared for this. With upcalls enabled, ASTs are delivered only when all the threads enable them.

Image exit

OpenVMS provides user mode exit handlers to be declared using $DCLEXH. These exit handlers are called when the image is terminated (via $EXIT or $FORCEX). When upcalls are disabled, the threads library is unaware of image exit being requested. If image exit begins with $EXIT, then the exit handlers are executed in the current thread. However, if the image exit begins with $FORCEX, then the exit handlers are executed in some random thread. As this random thread is time sliced, and other threads are also eligible to run, it imposes few limitations in the way exit handler could be written. For example, in the exit handler you need to be careful while calling pthread_join for a thread, because the target thread might be the thread that was interrupted. This could lead to a deadlock. Also, if the exit handler uses a lot of stack space, it could lead to a stack overflow.

If upcalls are enabled, the threads library creates an extra internal thread called "exit-handling thread." This thread is created during image startup, and is immediately put into blocked state and remains in that state until the normal execution of the program. If an image exit is requested, OpenVMS Exec informs the threads library via an exit upcall. The exit upcall wakes the exit-handling thread and executes the exit handlers in that thread. The exit handling thread executes at highest priority, but if it blocks (for example, pagefault, pthread_join etc), then other threads are eligible to run. If the image exit is induced by $EXIT, then the current thread ceases to exist. The exit handling thread is created only when upcalls are enabled, and is the only internal thread to execute user written code (exit handlers). Because the exit handling thread is an internally created thread, the stack for the thread is limited. If your application requires more stack for exit handlers you can define the following logical:

$ DEFINE PTHREAD_CONFIG "stksize=exit=200000"

The above PTHREAD_CONFIG logical setting results in 200,000 bytes of the stack being reserved for the exit handling thread. This feature is available from OpenVMS V8.4 onwards. (It is also available on OpenVMS V8.3-1H1 with VMS831H1I_PTHREAD-V0200 or above.)

Conclusion

Application performance varies depending on multiple factors as discussed in this paper. It is ideal to check your application with various combinations of the parameters to figure out what works best for the specific application. In addition to these recommendations, to get maximum performance benefit, the application has to be designed in such a way that it makes effective use of the available CPUs.

Resources

HP OpenVMS programming concepts manual:
http://h71000.www7.hp.com/doc/82final/5841/5841pro_005.html

HP OpenVMS systems ask the wizard:
http://h71000.www7.hp.com/wizard/wiz_1062.html

HP OpenVMS linker utility manual:
http://h71000.www7.hp.com/doc/83final/4548/4548pro_026.html

For more information

To know more details on HP POSIX Threads Library overview and programming guidelines, visit http://h71000.www7.hp.com/doc/73final/6493/6101pro_contents.html.

Share your feedback or queries on this HP OpenVMS technical journal here.

4AA4-4916ENW, January 2013