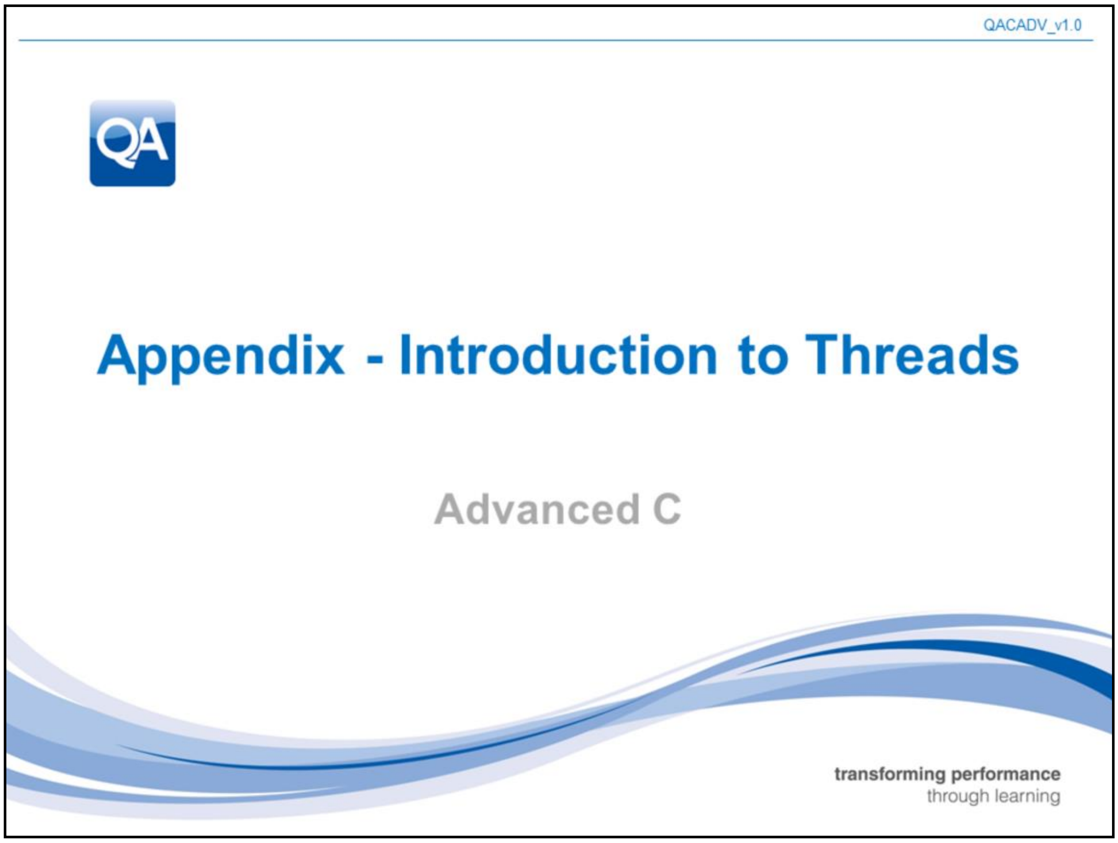
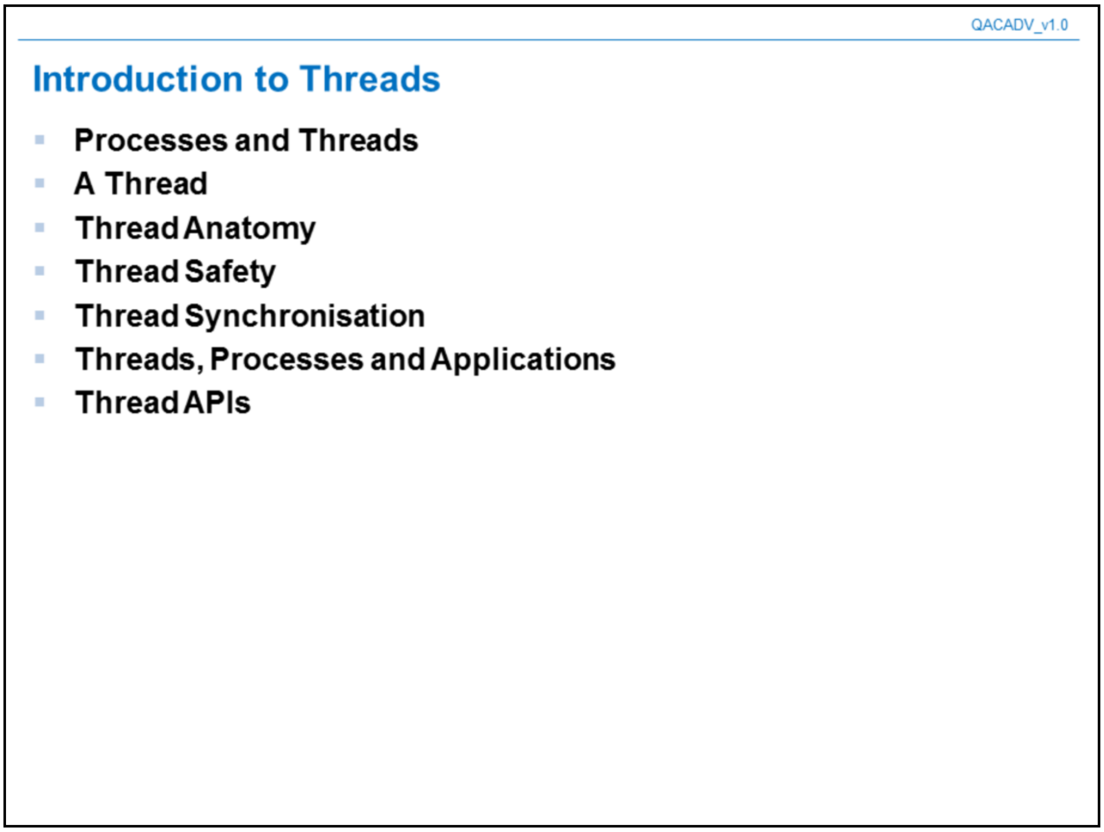


QACADV_v1.0

Appendix - Introduction to Threads

Advanced C

transforming performance
through learning

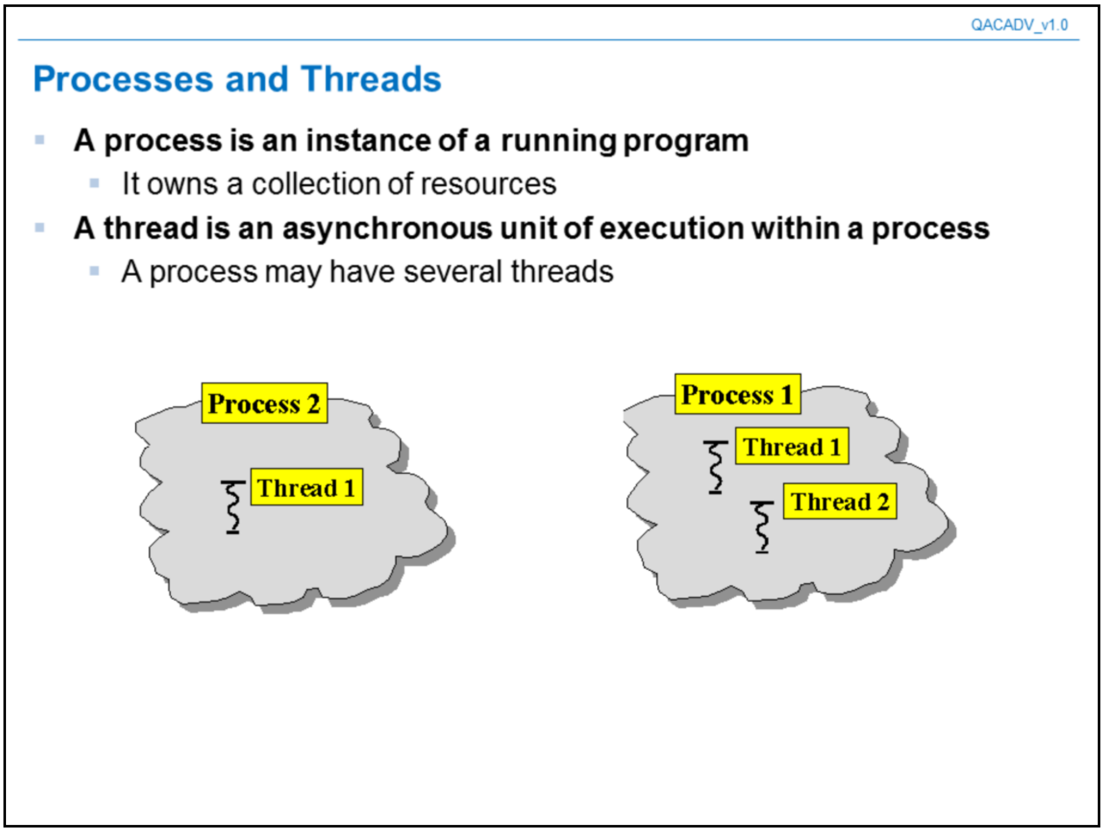


QACADV_v1.0

Introduction to Threads

- Processes and Threads
- A Thread
- Thread Anatomy
- Thread Safety
- Thread Synchronisation
- Threads, Processes and Applications
- Thread APIs

This section gives an overview of multi-thread programming, without detailing individual implementations, which vary considerable in their detail.



QACADV_v1.0

Processes and Threads

- **A process is an instance of a running program**
  - It owns a collection of resources
- **A thread is an asynchronous unit of execution within a process**
  - A process may have several threads

Process 2
Thread 1
Process 1
Thread 1
Thread 2

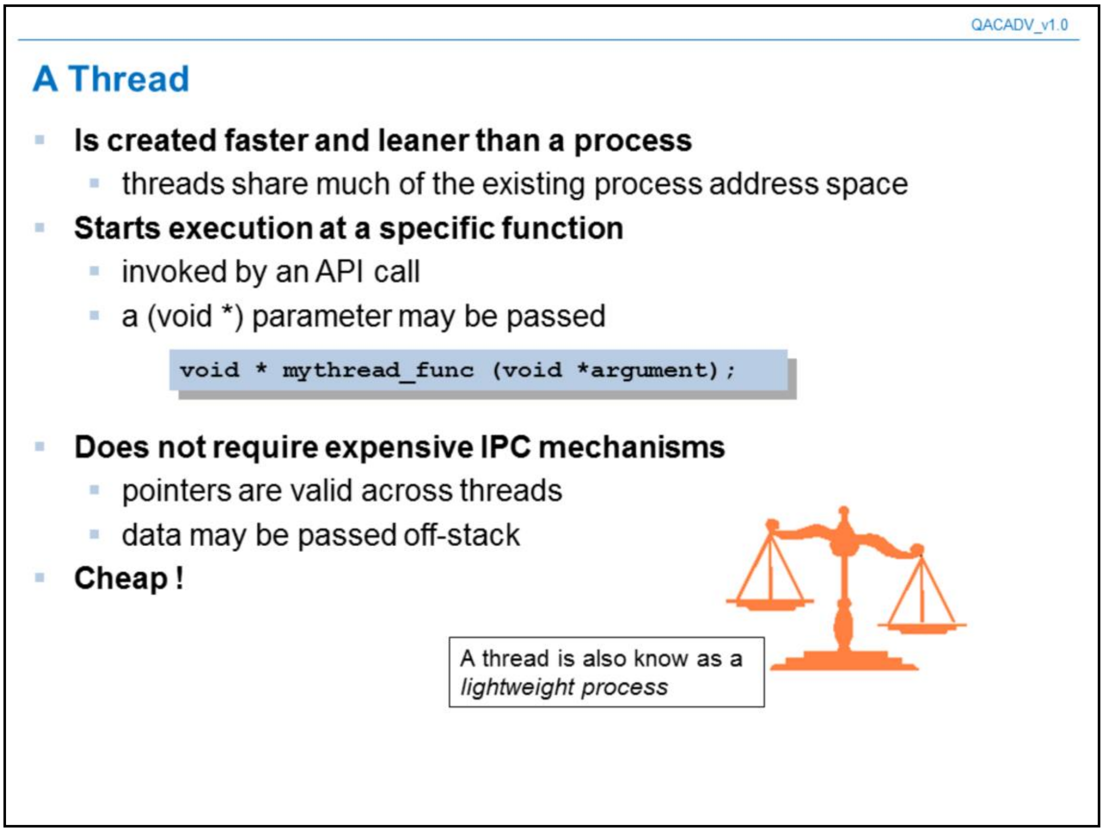
Modern general purpose operating systems, such as Unix and Microsoft Windows, allow a user to run several applications simultaneously, so that non-interactive tasks can be run in the background while the user continues with other work in the foreground. The user can also run multiple copies of the same program at the same time.

Less obviously, the facility for running several parts of a single application simultaneously allows the programmer to improve the performance of many applications by designing them so that individual programming tasks are carried out independently and in parallel (asynchronously), rather than in sequence.

A thread is a unit of execution within a process. Each thread has a function to execute, and a CPU register state and stacks to enable the operating system to preemptively schedule them. Processes can contain several threads. Threads are used by the programmer to perform asynchronous subtasks that cooperate towards a common goal, their combined effect being the purpose of the process to which they belong.

Operating systems designed around threads, such as Windows 2000 and XP, see a process as merely a container for threads.

Some operating systems schedule threads in the same way as processes, which can lead to unfair scheduling. For example, a process with four threads could get four times as much CPU as a process with only one thread.



QACADV_v1.0

A Thread

- **Is created faster and leaner than a process**
  - threads share much of the existing process address space
- **Starts execution at a specific function**
  - invoked by an API call
  - a (void *) parameter may be passed

```
void * mythread_func (void *argument);
```

- **Does not require expensive IPC mechanisms**
  - pointers are valid across threads
  - data may be passed off-stack
- **Cheap !**

A thread is also know as a *lightweight process*

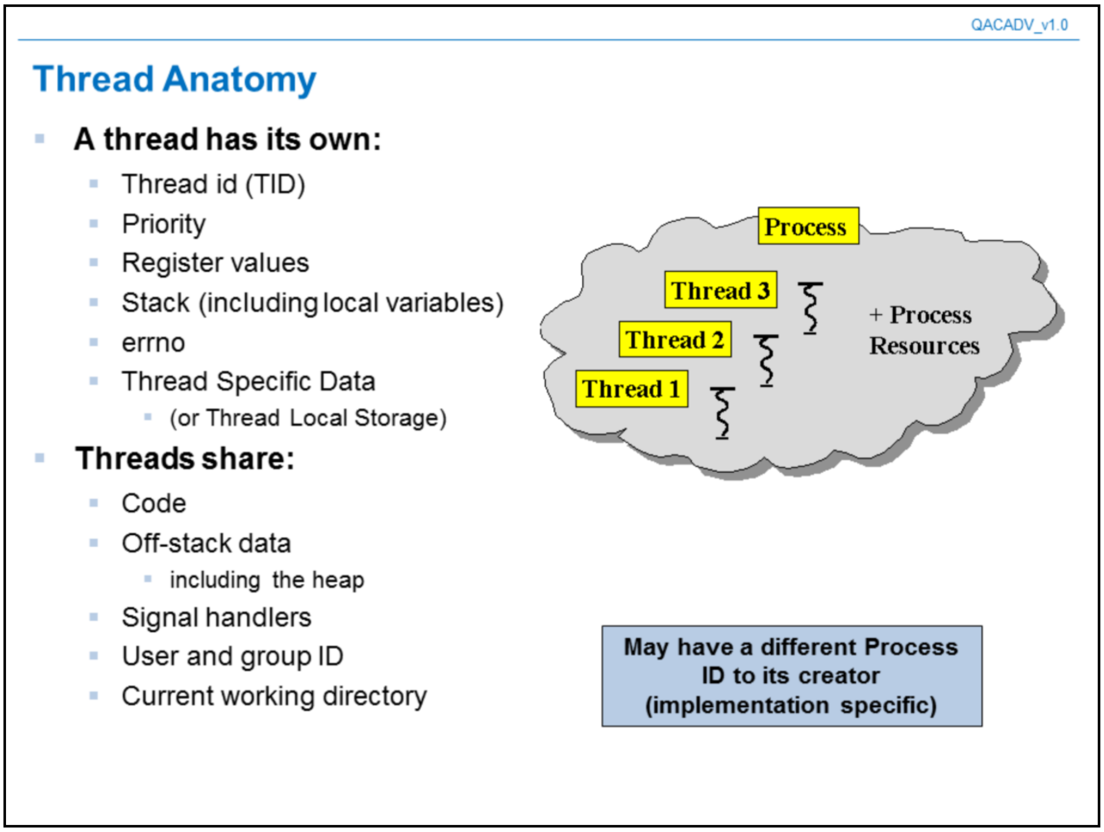
There are various strategies for creating a process. Microsoft Windows create a new *process address space*, map the specified program to it, then start it running. Unix creates processes using `fork()`, which is relatively fast. With `fork()`, components of the parent's process address space are copied to make up the new process. The child process starts its execution in the same function as its parent, immediately after the return from `fork()`. This is often followed by a call to one of the *`exec`* functions to map a new executable.

Most operating systems also use the *copy-on-write* method to speed this further. With copy-on-write, pages are not physically copied unless and until they are written to (this is how code pages are shared).

Even so this is still not fast enough for some applications, and because pages are often written to, can be heavy on memory resources. With threads a large proportion of a process's pages are physically shared - even when they are written to, and large data areas do not need to be initialised. This makes thread creation fast (10-100 times faster) and, once running, very *lightweight*.

Passing data between processes requires IPC mechanisms like shared memory, memory-mapped files, or message queues. Passing data between threads could not be easier. Because they share the same process address space, pointers passed between threads are valid, and off-stack (global) items may be used for the transfer. The heap is still shared, so a pointer to an area `malloc`'ed by one thread may be used by another.

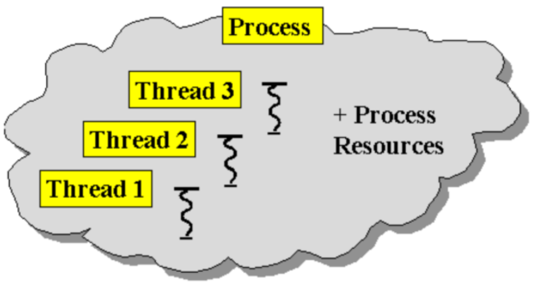
This can be something of a double-edged sword! We still need a synchronisation mechanism, and programmers can no longer write to global areas without a thought. There is no avoiding the fact that writing multithreaded code is *different*!



QACADV_v1.0

Thread Anatomy

- **A thread has its own:**
  - Thread id (TID)
  - Priority
  - Register values
  - Stack (including local variables)
  - errno
  - Thread Specific Data
    - (or Thread Local Storage)
- **Threads share:**
  - Code
  - Off-stack data
    - including the heap
  - Signal handlers
  - User and group ID
  - Current working directory



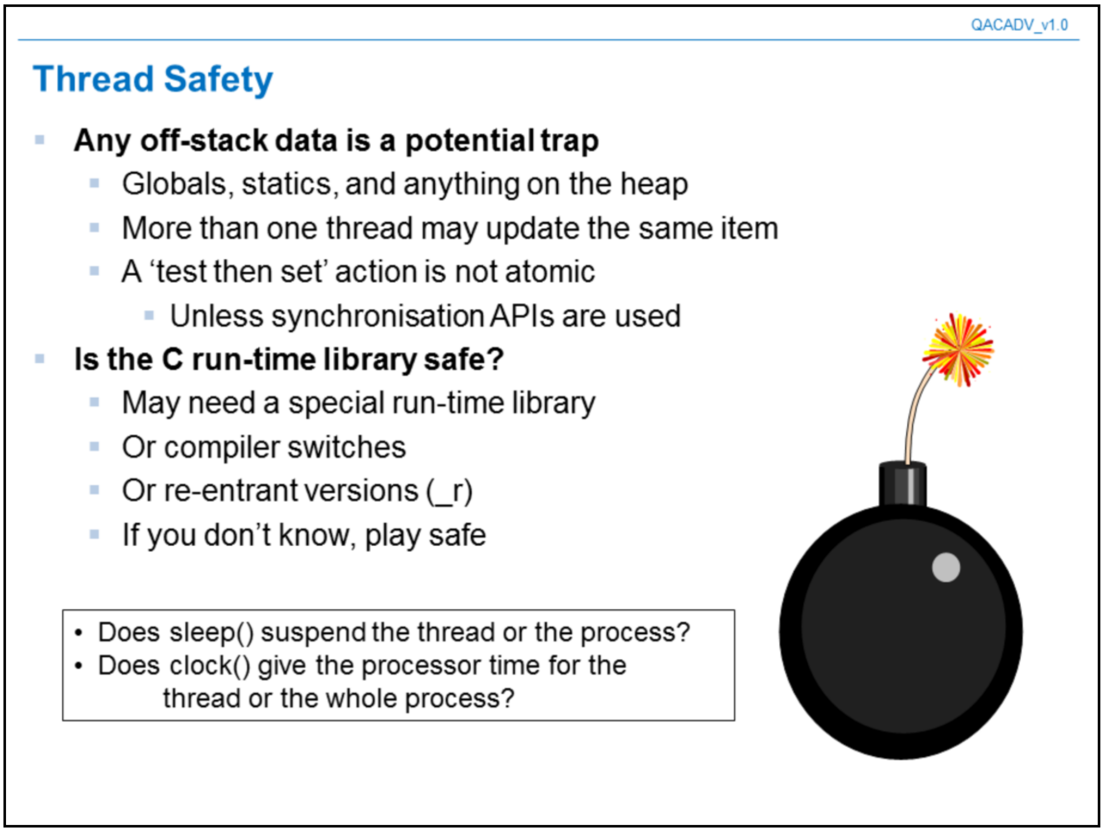
May have a different Process ID to its creator (implementation specific)

Threads provide a mechanism for carrying out several programming tasks simultaneously within a process. Each thread runs independently and maintains a set of data for saving its context while waiting to be scheduled for processing time. These structures include the thread's own set of machine registers, its own stack, and a private memory block.

From a programming language perspective, a thread may be considered an asynchronous function. When a normal C function is called, the flow of execution transfers from the calling environment to the function, and then returns when the function ends. But when one thread starts another, the original continues to run while the new thread executes concurrently and independently. There may be many copies of a function running within a process as different threads. Threads of the same process can execute any part of the program's code, including a part being executed by another thread.

Creating a new thread is 'cheap and fast' in comparison to a process, in terms of system overhead and time to initialize internal data structures. Global resources of the process are shared by its threads and so they communicate simply, but they are not protected from each other. The programmer often needs to serialize access to data using synchronization objects to coordinate their activities.

When designing your application you should remember the occasions when threads can be terminated. Any thread can call `exit()`, which will kill all the other threads regardless of what they are doing at the time. To avoid this you can use cancellation points (POSIX) or control the whole lot by waiting on the threads to complete. A robust architecture is where `main()` starts the "worker" threads, then waits on each to complete before exiting. No other thread should ever use `exit()`.



QACADV_v1.0

Thread Safety

- **Any off-stack data is a potential trap**
  - Globals, statics, and anything on the heap
  - More than one thread may update the same item
  - A 'test then set' action is not atomic
    - Unless synchronisation APIs are used
- **Is the C run-time library safe?**
  - May need a special run-time library
  - Or compiler switches
  - Or re-entrant versions (_r)
  - If you don't know, play safe

- Does sleep() suspend the thread or the process?
- Does clock() give the processor time for the thread or the whole process?

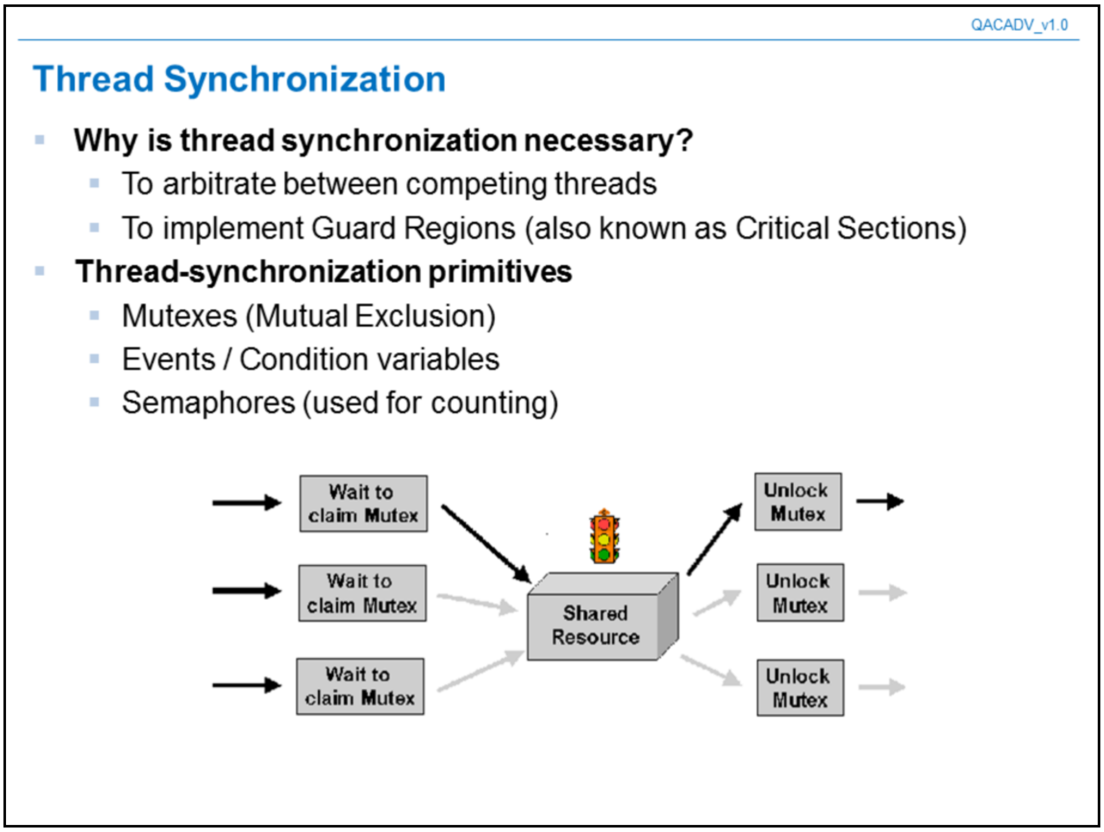
Potentially, anything which is shared between threads can cause contention. A global variable may be tested, then updated depending on its value. The problem is that this action is not atomic - it can be interrupted between the test and the update.

Often code is written to store progress information in globals, again this may not be thread-safe since different threads may be progressing at different rates. The C run-time library function `strtok()` is a good example of this.

Imagine the havoc to the heap pointers if two threads call `malloc()` at exactly the same time! Fortunately the Posix.1 standard insists that `malloc()` and `free()` are thread-safe, together with the `stdio` library and some critical globals, like `errno`. However some compilers require _REENTRANT to be defined to enforce this. Some library routines cannot enforce thread-safe versions, but may have special "re-entrant" versions of them, which may be recognised with the `_r` suffix on their names. Examples are `readdir_r()`, `asctime_r()`, and `strtok_r()`. Note that they often have an additional parameters. Microsoft Windows provides special multithreaded C run-time libraries instead.

Other functions have special effects, for example will `sleep()` cause the entire process to suspend, or just the calling thread? Some functionality just will not mix with a multithreaded architecture, signals are an example.

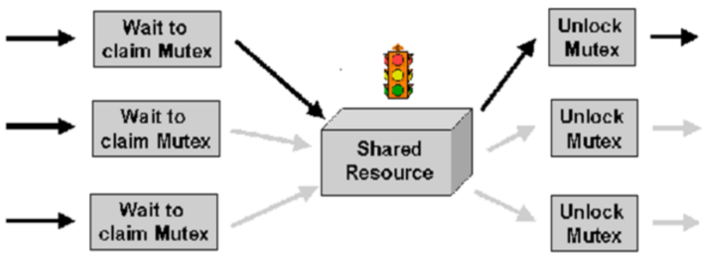
Thread safety is a particular issue when calling third party libraries. The only way of knowing which functions are thread-safe is to ask your supplier. It may be necessary to implement your own synchronisation using, for example, a *mutex* (mutual exclusion flag). Of course this could end up serialising everything, which defeats the object of using threads!



QACADV_v1.0

Thread Synchronization

- **Why is thread synchronization necessary?**
  - To arbitrate between competing threads
  - To implement Guard Regions (also known as Critical Sections)
- **Thread-synchronization primitives**
  - Mutexes (Mutual Exclusion)
  - Events / Condition variables
  - Semaphores (used for counting)

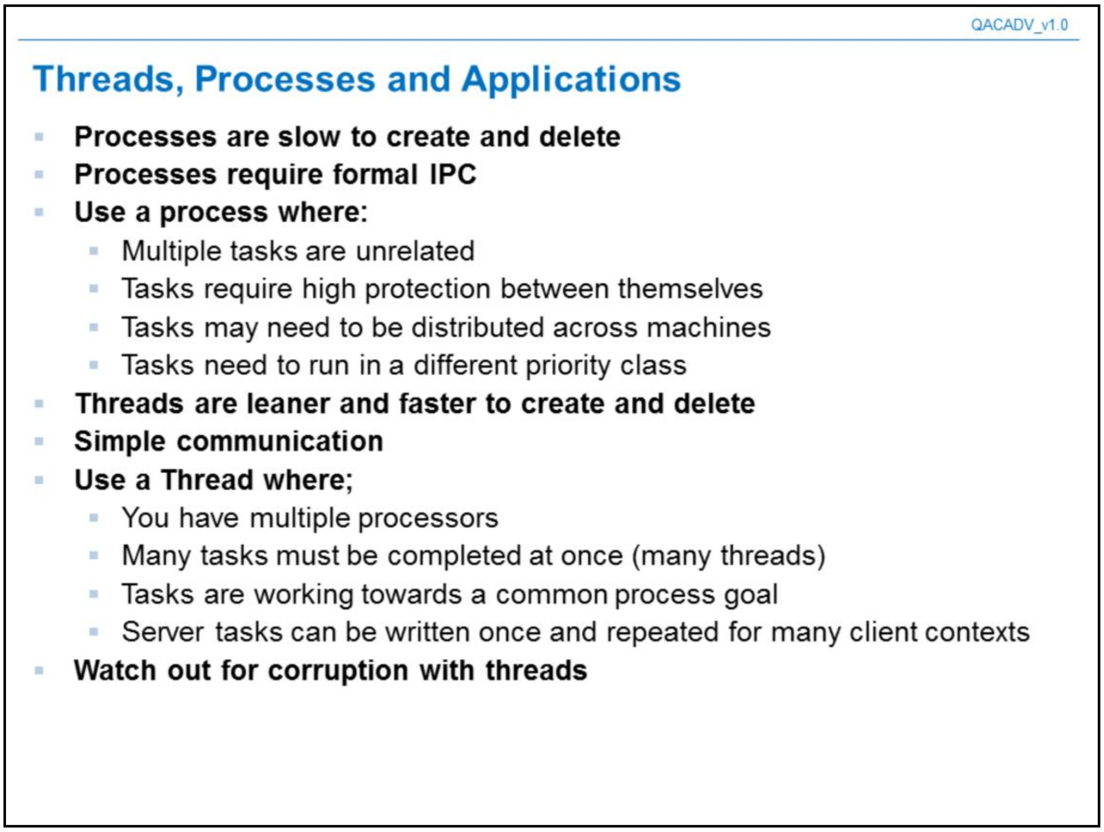


In a multitasking environment, it is essential to coordinate the execution of multiple threads in one or more processes.

The scheduler may preempt a thread at any time, including while it is in the middle of accessing a data area, device, or section of non-reentrant code. Such '*serially reusable resources*' should only be used by one thread at a time, otherwise their data may become corrupted or a deadlock may occur. Also, it is sometimes important to coordinate threads; one thread may need to wait for another to complete an action before carrying out its task. The work of ensuring that only one thread at a time accesses a shared resource is known as '*arbitration*' or '*mutual exclusion*', while the coordination of threads is referred to as '*synchronization*'.

Synchronization objects are essentially flags maintained by the operating system, which enable threads to signal each other in order to synchronize their activities and to protect non-reentrant code and resources by providing mutual exclusion.

Many thread libraries include a "lightweight" mutex, which executes within the process address space, and thus may be used for synchronizing threads, but not processes. This gives a considerable performance advantage, since a kernel call is not required to service the mutex. On Microsoft Windows these are called *Critical Sections*.



QACADV_v1.0

Threads, Processes and Applications

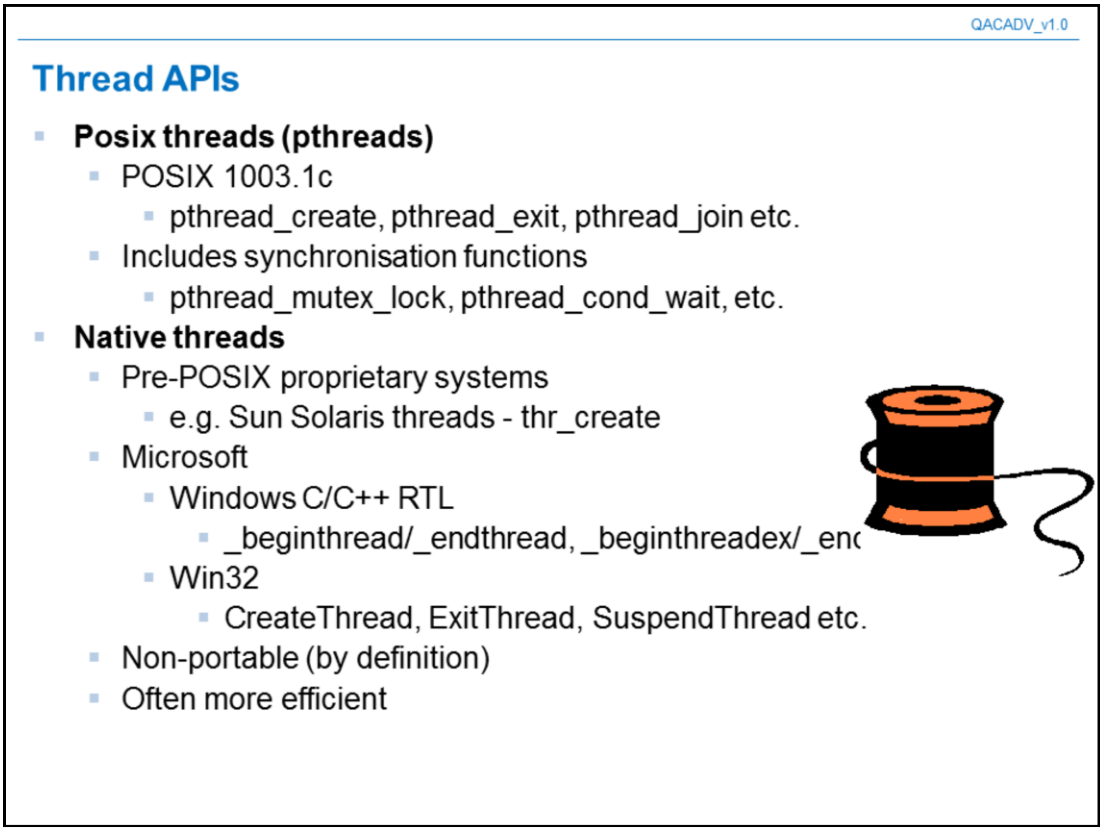
- **Processes are slow to create and delete**
- **Processes require formal IPC**
- **Use a process where:**
  - Multiple tasks are unrelated
  - Tasks require high protection between themselves
  - Tasks may need to be distributed across machines
  - Tasks need to run in a different priority class
- **Threads are leaner and faster to create and delete**
- **Simple communication**
- **Use a Thread where;**
  - You have multiple processors
  - Many tasks must be completed at once (many threads)
  - Tasks are working towards a common process goal
  - Server tasks can be written once and repeated for many client contexts
- **Watch out for corruption with threads**

A process may contain one or more threads, and can create and terminate other processes. An application may therefore use multiple processes, each containing multiple threads.

The design considerations for an application are summarized above. Unlike threads, processes offer a protected environment, but take much longer to create than threads. It is easier and quicker to share data between threads within a process, rather than to share data between processes by any of the means of IPC. Processes are therefore best used for major, functionally separate, sections of an application that are not constantly invoked.

A process based system is more resilient than one based on threads. A wayward pointer in a single-thread process will only affect that one thread, but in a multi-threaded application anyone could be toasted - and tracking the error can be difficult. Many debuggers cannot handle multi-threaded applications, in particular they can have difficulty in interpreting a core file (Unix).

The actual implementation of threads at the kernel level varies between implementations. On Linux, for example, the kernel knows nothing about threads. The implementation actually uses one kernel context for each thread. That means that threads are *not* a way of by-passing kernel limits on the number of processes, that limit will actually restrict the number of threads as well. Windows NT / 2000 / XP, on the other hand, is specifically designed around threads rather than processes.



QACADV_v1.0

Thread APIs

- **Posix threads (pthreads)**
  - POSIX 1003.1c
    - pthread_create, pthread_exit, pthread_join etc.
  - Includes synchronisation functions
    - pthread_mutex_lock, pthread_cond_wait, etc.
- **Native threads**
  - Pre-POSIX proprietary systems
    - e.g. Sun Solaris threads - thr_create
  - Microsoft
    - Windows C/C++ RTL
      - _beginthread/_endthread, _beginthreadex/_enc
    - Win32
      - CreateThread, ExitThread, SuspendThread etc.
  - Non-portable (by definition)
  - Often more efficient

Most Unix implementations will follow the POSIX 1003.1c-1995 standard, but you should check with your vendor which standard your version supports. For example, HP-UX releases prior to Release 11.0 complied only with the older POSIX 1003.4a draft 4 standard. Tru64 offers both, depending on which library you link with.

The X/Open equivalent, called Unix 98, is a superset of POSIX 1003.1c (so far as threads are concerned). Unix 98 also includes pthread based read-write file locks, however these are not yet part of the POSIX standard (working group 1003.1j).

Threads have been around for a long time, and many implementation pre-date POSIX threads. Their function calls are not portable, and are not compatible with pthreads. Since they are written for a specific architecture, then native threads are almost always going to be more efficient than pthreads. The decision has to be made on portability grounds, ease of maintenance, and performance.

Programmers used to Win32 multi-threading can quickly pick up pthreads, though they will miss WaitForMultipleObjects. Similarly, porting a multi-threaded design to Windows should not be a big problem, the concepts are similar. It should be noted that the Microsoft Win32 API's CreateThread/ExitThread are not suitable for use with C, and the C run-time library specific functions _beginthreadex/_endthreadex should be used instead.