How does it work?#

Being a microkernel OS, most of the "operating system" in QNX is implemented in user processes. Even the process manager is implemented as a multithreaded process that uses the regular QNX message passing and thread support (the only difference is that is allowed to perform certain privileged operations not available to normal processes).

This means that many of the tricky issues for SMP operation (synchronisation, mutual exclusion etc.) are already dealt with by the POSIX pthread interfaces. Adding more cpus to the system simply gives the opportunity for more than one thread to run at a time.

The tricky parts of the SMP implementation are thus limited to the microkernel, memmgr and the startup support it uses to initialise the system and implement (board-specific) inter-processor communication. This adds a certain amount of complication and overhead that is really only needed when the system has multiple cpus, so as an optimisation, there are two different procnto variants:

- procnto only implements support for a uniprocessor
- procnto-smp implements support for one or more processors

The extra code for SMP operation is in:

- VARIANT_smp conditional code
- smp directories that contain code to override uniprocessor implementations:

SMP and mutual exclusion of kernel data structures#

The kernel only allows one CPU into its code on an SMP system. This is normally OK, because we're not supposed spend a lot of time processing each kernel call. The kernel never blocks itself waiting for something - a user thread might block waiting for a resource, but the kernel itself does not. There's just one small gotcha - message passing. If the kernel gets asked to copy 10 meg of data, it's going to take a while. The solution for that is the SMP_MSGOPT compile define. When that gets turned on (currently only for X86 SMP), the kernel will look at the size of the message and, if it's large enough, it will 'give up' the lock that keeps other CPU's out of the kernel while it's doing the big copy. That will allow other CPU's to complete their kernel calls while the copy is going on. Once the CPU has finished the message pass copy, it will re-acquire the kernel lock and the complete the remaining portion of the send/receive/reply call. There's no particular reason that we've only done this for the X86 SMP - we just haven't had time to propagate the CPU specific changes required for SMP_MSGOPT to the other architectures.

Note that just because the code only allows one CPU into the kernel at a time, that doesn't mean that multiple things aren't happening in the procnto code at the same time. One CPU might be in the kernel and another might be handling a hardware interrupt - we need don't 'enter' the kernel to do that. Additionally, the process manager (everything in **services/system** outside of the **ker** subtree) is running as a normal process as far as the kernel is concerned, so multiple CPU's could be running different threads from it at the same time. Process manager code just uses the normal process synchronization primitives (mutexes, condvars, etc) to protect its data structures).

Details#

The following sections delve into the details of:

- how the system initialises all the cpus
- the startup support required
- SMP kernel code

Background articles:

- SMP chapter of the System Architecture Guide. (ps. Marketing likes to call SMP "Multicore".)
- SMP: Two Processors and Beyond Basic intro to the effects of SMP on application programming.