A Brief Introduction to QNX

Dean	1930026019
Lydia	1930026132
Sarah	1930026051

Content

What is QNX?

As the first successful commercial microkernel operating system, QNX is a distributed, embedded, scalable hard RTOS owned by Blackberry Limited. It follows POSIX.1 (program interfaces) and POSIX.2 (shells and tools), and partially follows POSIX.1b (real-time extensions). It was born in 1980 and is now 40 years old. Starting in 2020, It is widely applied in various devices, including automobiles and cell phones. To date, QNX has a nearly 75 percent market share in the automotive market, with QNX-based systems used by nearly every major global automotive brand, including Alfa Romeo, Jeep, Audi, BMW, Honda, Buick, etc. In addition to the automotive, QNX's biggest order comes from Cisco, which uses the operating system for almost all the mid-to-high end routing devices, therefore, it makes networking the second largest application area of QNX. In addition, QNX works closely with big companies such as General Electric, Alstom, Siemens, Lockheed Martin, and NASA, and plays an active role in rail transportation, medical devices, smart grids, and aerospace.

BlackBerry Limited

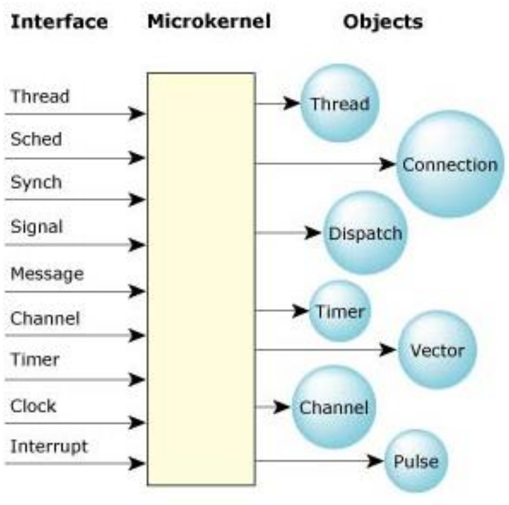
The History of QNX

Dan Dodge, the QNX creator

The origin of QNX is quite interesting, it came from an Operating System course. Dan Dodge and Gordon Bell, who were all students of the University of Waterloo in 1980, attended a session on the topic of real-time operating systems, in this class the students had to build a fundamental real-time microkernel program for users. They all saw a commercial need for such a system, and in that year they moved to Kanata, a high-tech planning community in Ontario, and established a corporation called Quantum Software Systems in 1980. Then the initial edition of QUNIX was published in 1982, and in 1984, the name of QUNIX was changed to QNX by Quantum Software Systems to prevent copyright disputes, and in 2010, The corporation was eventually obtained by BlackBerry Ltd.. The latest version of QNX, QNX Neutrino has been migrated to many platforms and is now available on virtually all modern CPU series used in the embedded computing market, including the most mainstream PowerPC, x86, MIPS, SH-4, and the closely related ARM, StrongARM, and XScale systems.

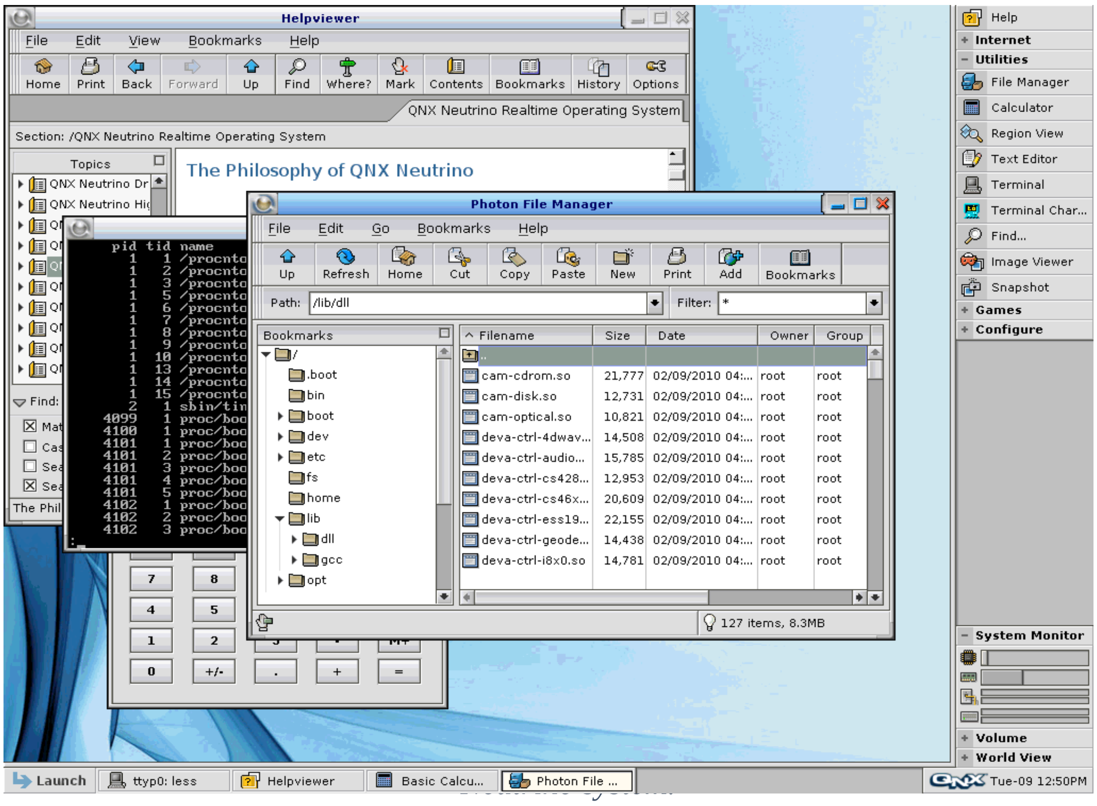
How is QNX designed?

As a microkernel-based OS, the idea of QNX is to run most of the OS kernel as several tiny tasks called Resource Managers. the QNX kernel (procnto) enables only the core POSIX functions used in embedded real-time systems, including processor scheduling, inter-process communication, interrupt redirection, and timers. Anything else runs as user processes, including a special process called proc, which provides process creation and memory management by running alongside the microkernel. This is unlike the other more conventional monolithic kernels, where the operating system kernel is a remarkably large-scale program consisting of many components with specific functions. With respect to QNX, the microkernel allows developers and users to block any functionality they do not need without having to modify the OS.



The QNX Neutrino microkernel.

This is achieved through two critical mechanisms of QNX: the inter-process communication of the subroutine call type, and the bootloader that allows loading an image containing the kernel and any necessary assemblies and shared libraries. No device drivers are available in the kernel. The networking stack is built upon the NetBSD code. In addition to supporting its original local device drivers, QNX also offers backup support for its network drivers and traditional io-net manager server migrated from NetBSD. The operating system is built on this foundation.



```
    pid tid name
      1   1 /procnto
      1   2 /procnto
      1   3 /procnto
      1   5 /procnto
      1   6 /procnto
      1   7 /procnto
      1   8 /procnto
      1   9 /procnto
      1  10 /procnto
      1  13 /procnto
      1  14 /procnto
      1  15 /procnto
      2   1 sbin/tin
   4099   1 proc/boo
   4100   1 proc/boo
   4101   1 proc/boo
   4101   2 proc/boo
   4101   3 proc/boo
   4101   4 proc/boo
   4101   5 proc/boo
   4102   1 proc/boo
   4102   2 proc/boo
   4102   3 proc/boo
:
```

Filename	Size	Date	Owner	Group
..				
cam-cdrom.so	21,777	02/09/2010 04:...	root	root
cam-disk.so	12,731	02/09/2010 04:...	root	root
cam-optical.so	10,821	02/09/2010 04:...	root	root
deva-ctrl-4dwav...	14,508	02/09/2010 04:...	root	root
deva-ctrl-audio...	15,785	02/09/2010 04:...	root	root
deva-ctrl-cs428...	12,953	02/09/2010 04:...	root	root
deva-ctrl-cs46x...	20,609	02/09/2010 04:...	root	root
deva-ctrl-ess19...	22,155	02/09/2010 04:...	root	root
deva-ctrl-geode...	14,438	02/09/2010 04:...	root	root
deva-ctrl-i8x0.so	14,781	02/09/2010 04:...	root	root

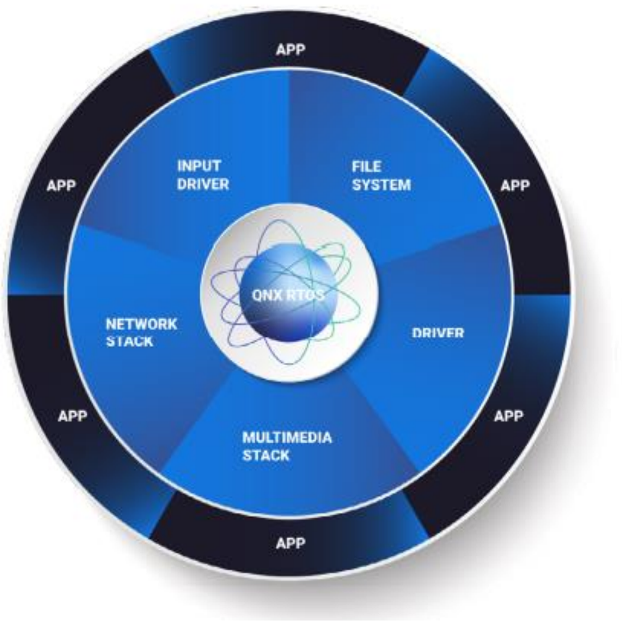
Introduction of QNX Neutrino Microkernel

The most famous feature of the QNX operating system is its design of a microkernel named Neutrino RTOS.

The main features of POSIX that are used in real-time systems are implemented in the Neutrino microkernel, additionally with its own message-passing services.

The microkernel has kernel calls to support the following:

1. Thread
2. Message passing
3. Signals
4. Clocks
5. Timers
6. Interrupt handlers
7. Semaphores
8. Mutual exclusion locks
9. Condition variables
10. Barriers [1]



As for its architecture, Neutrino chooses to isolate every non-kernel part outside the kernel and gives each part its own address space. (Parts include: Application, Driver, Protocol stack, Filesystem) This characteristic may help the kernel be more secure and efficient when facing the failure of any components. This means that kernel and other components will never be bothered (or taken down) while one of the components failed, it can immediately restart without big negative influences to others, thus minimizing the possibility of breaking down the whole system.

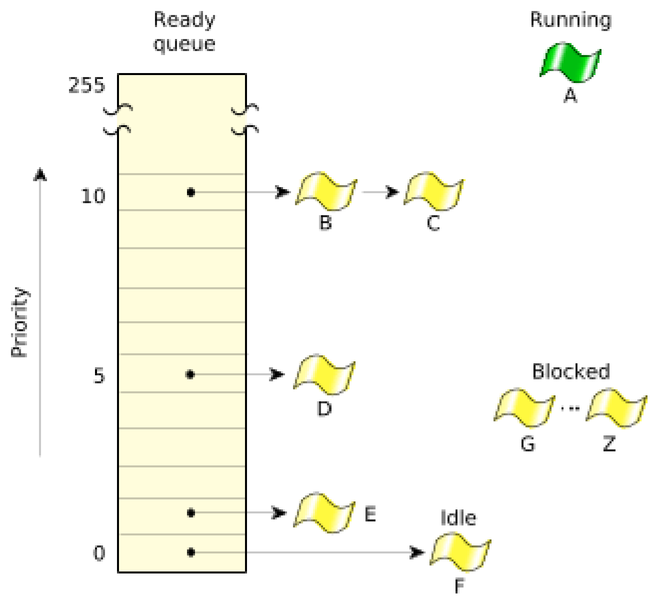
Thread scheduling algorithm in Neutrino Microkernel

- ### Scheduling decision

The scheduling decision is made whenever the execution state of any thread changes, the scheduler will perform a context switch when the running thread: blocks, preempted, or yields. [2]Block happens when the running thread must wait for some event to occur. The scheduler will only focus on state running and ready,

[1] Reference from BlackBerry QNX user guide.

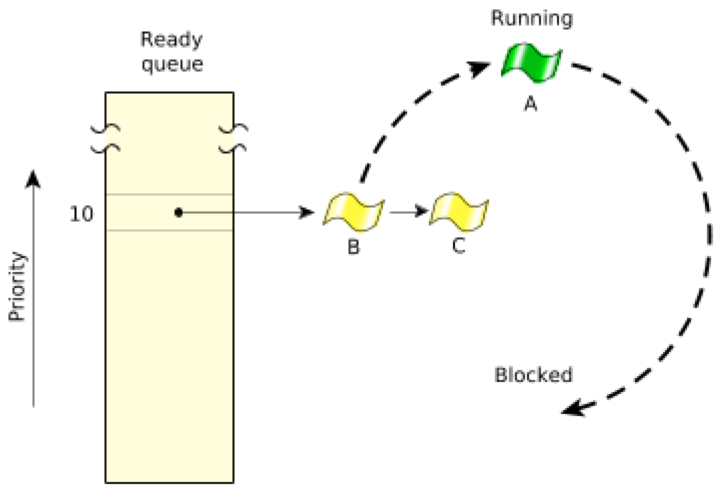
the block will be ignored by the scheduler.



The ready queue in a single-core system

- **Priority levels in ready queue**

The ready queue can be divided into 256 scheduling priority levels, bigger number represents the higher priority. The running scheduling is based on their priority, FIFO and round-robin scheduling will only be applied when there are two or more threads that share the exact same priority in the list additionally in the ready queue.



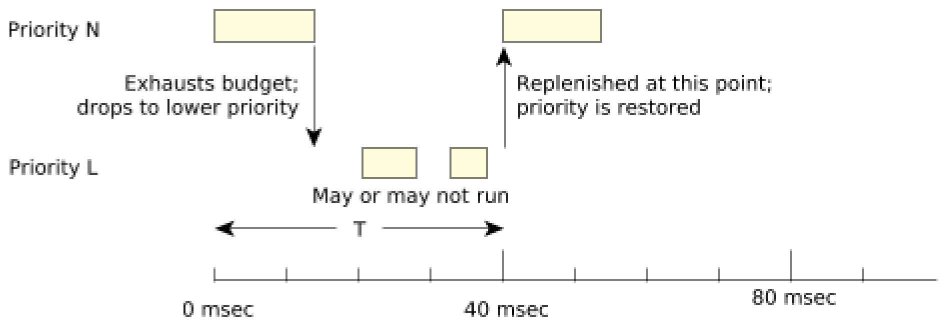
When two threads share the same priority

For example, the case that shown above. When A is blocked, B and C are with the highest and same priority in ready queue, the next thread to be executed should be B since B came first, it used FIFO/round-robin, then C will get the turn, and so on.

- **Scheduling Algorithm**

  - **Sporadic Algorithm**

Sporadic Algorithm used to provide a capped limit on the execution time of a thread within a given period of time, especially the thread that will run for a long time.



Priority N
Exhausts budget; drops to lower priority
Replenished at this point; priority is restored
Priority L
May or may not run
T
0 msec
40 msec
80 msec

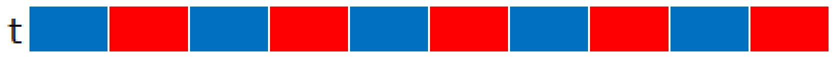
Sporadic Algorithm

This algorithm considers two levels of priority --- foreground, and background. Foreground has higher priority than background, if a high priority thread1 continuously occupies the CPU for a certain time, we called the thread exhausts the budget that system gave it, thread1 will be forced to drop the priority to background. If there is another thread2 that has a higher priority than background, then thread2 will be executed, otherwise, thread1 can continue. After waiting for a while, also the specific time period that system will determine, thread1 will be brought back to foreground and run again till it exhausts the budget or finishes execution.

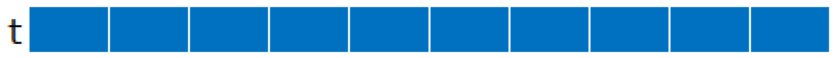
  - **Adaptive partition scheduling**

As we mentioned previously, when discussing the priority scheduling, we only discuss that when there are few threads which have the same priority, how do system do. When the priority is different, the thread which have higher priority wins. However, “how much higher” is out of question. For example, two threads with priority 10 and 11 have the same situation with another two threads with priority 10 and 30.

Assume we have two threads, represented by red and blue, they have the same priority, the scheduling method is RR, then in the 10-time slice we can get the following result:

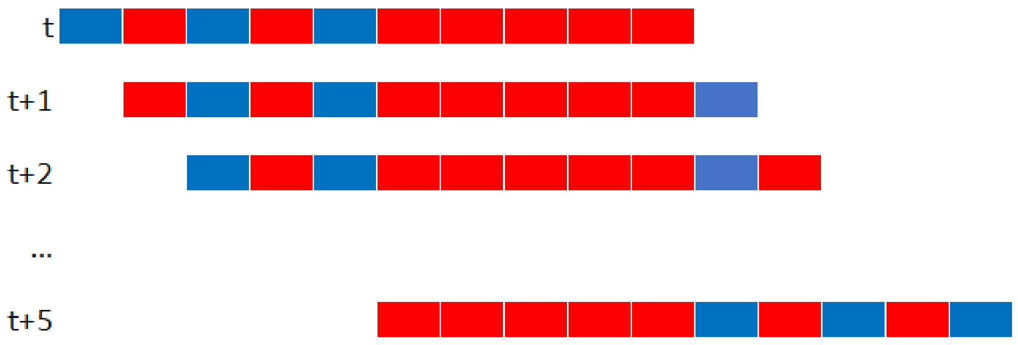


However, if we let the priority of blue thread rise even 1, the result will show as following:



This situation may make system not perform optimally.

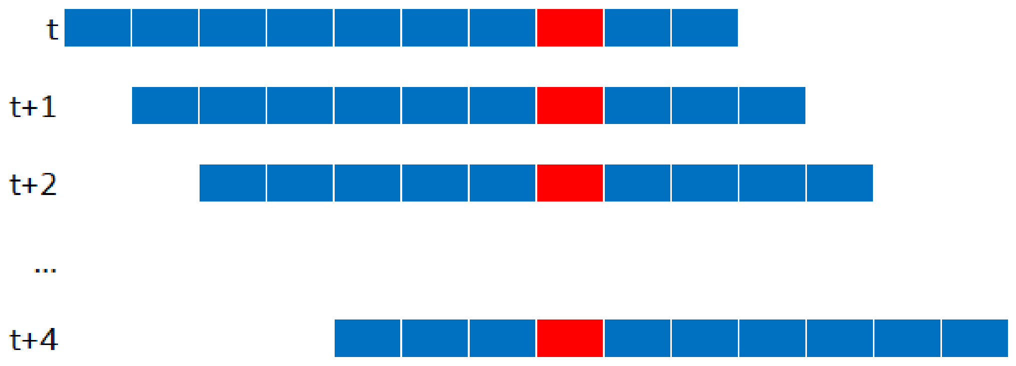
Luckily, there is a method called "partition scheduling". The idea is to divide the CPU computational ability into few sections, for example 70% and 30%, then put different threads into these sections. When the CPU budget in the section is used up, then all the executable thread in the corresponding section will be stopped until the budget recover. If we put the red thread into the 70% section, blue thread into the 30% section, 10-time slice, then the threads will occupy the CPU as following:



partition scheduling

Notice that in the 7th time slice, although the two threads have the same priority, it seems that it is the tern for the blue thread. However, the blue thread has used up the three time slices of ten, so the system doesn't execute the blue thread, but continue let red thread occupy the CPU.

In addition to the traditional partition scheduling, QNX add the part of "self-". The basic idea is the same, divide the CPU computational ability, then put different threads into different sections. But the special of QNX is, when any section is rich in CPU computational ability, self-adaption algorithm allows the system borrow the extra computational ability to those section which need more computational ability.



"self-adaption" algorithm

As the graph shows, when the blue thread consumed all the computational budget in its own section, self-adaption algorithm will borrow its extra computational ability to the blue thread. In the 8th time slice, red thread needs to use CPU, thus blue thread make way at once.

○ Key thread

In reality, there have some important tasks need to be responded. For example, a emergency interrupt service thread need to be responded, whether the section have budget or not, it need respond. To solve this situation, in QNX self-adaption partition algorithm, allow the user distribute "key respond time", and set the specific thread as the "key thread".

If the section where the key thread in has budget, then the scheduling is as usual, however, if the section doesn't have the budget, then the key thread is allowed to "break" the section's budget, use the "critical respond budget" to execute.

Of course, a system shouldn't have too much key thread, if all the threads are the key thread, then the whole system is actually a normal priority scheduling system, all the sections and budgets are useless.

Conclusion

QNX's on-board operating system is now occupying more than 50% market. It has high stability and security. By analyzing the system architecture of QNX microkernel, the result shows that in addition to provide a compatible API for applications, microkernel is able to perform better than the single kernel system, besides, it can provide more powerful functions. The applications' source code remains the same, while expanding and developing the operating system become much more convenient. A flexible OS platform also offer more opportunities for the experiments such as replace a feature of operating system. In conclusion, microkernel operating system architecture started a new performance and a new standard of the functions.

Bibliography

1. BlackBerry QNX (2021). *Thread scheduling*. http://www.qnx.com/developers/docs/7.1/index.html#com.qnx.doc.neutrino.sys_arch/topic/kernel_SCHEDULING.html
2. 知乎. *QNX 的调度算法*. *https://zhuanlan.zhihu.com/p/300692946*
3. BlackBerry QNX (2007). *The QNX Neutrino RTOS 7.1*. https://blackberry.qnx.com/en/software-solutions/embedded-software/qnx-neutrino-rtos
4. Alchetron (2018). *Dan Dodge*. https://alchetron.com/Dan-Dodge
5. 10 Steps to Developing a QNX Program: Quickstart Guide. https://www.qnx.com/developers/docs/6.5.0SP1.update/com.qnx.doc.momentics_quickstart/about.html
6. QNX help document. http://www.qnx.com/developers/docs/6.5.0/index.jsp?topic=%2Fcom.qnx.doc.neutrino_sys_arch%2Fintro.html