Teaching material for RTOS

Dániel Drótos

Contents

Part I

Real Time Operating Systems
QNX Opearting System

Chapter 1

What is the QNX

QNX is an operating system for PCs. It has microkernel architecture and it is designed especialy for real time applications.

1.1 QNX's microkernel architecture

QNX's kernel (cf. Section 2) is very small -less then 8K. It's essentilal functions are:

1. **Message passing**. The kernel handles the routing of all messages among all processes throughout the entire system.
2. **Scheduling (cf. Section 7)**. The scheduler is a part of the Kernel, and is invoked whenever a process changes state as the result of a message or interrupt.

Unlike the processes, the Kernel itself is never scheduled for execution. It is entered only as the direct result of kernel calls, either from a process or from a hardware interrupt. All QNX services, exept those provided by the Kernel, are handled via standard QNX processes. A typical QNX configuration has the following system processes:

- Process manager (cf. Section 8) (**Proc**).
- Filesystem manager (cf. Section 10) (**Fsys**).

- Device manager (cf. Section 11) (**Dev**).
- Network manager (cf. Section 12) (**Net**).

1.2 Interprocess communication

QNX provides a simple but powerful set of Interprocess Communication (IPC) capabilities that simplify the job of developing applications made up of cooperating processes. In QNX, a message is a packet of bytes passed from one process to another. QNX attaches no special meaning to the content of a message.

Message passing not only allows processes to pass data to each other, but also provides syncronizing the execuion of several processes. As they send, receive, and reply to messages, processes go through various "changes of state" that affect when, and for how long, they may run.

1.3 QNX as a network

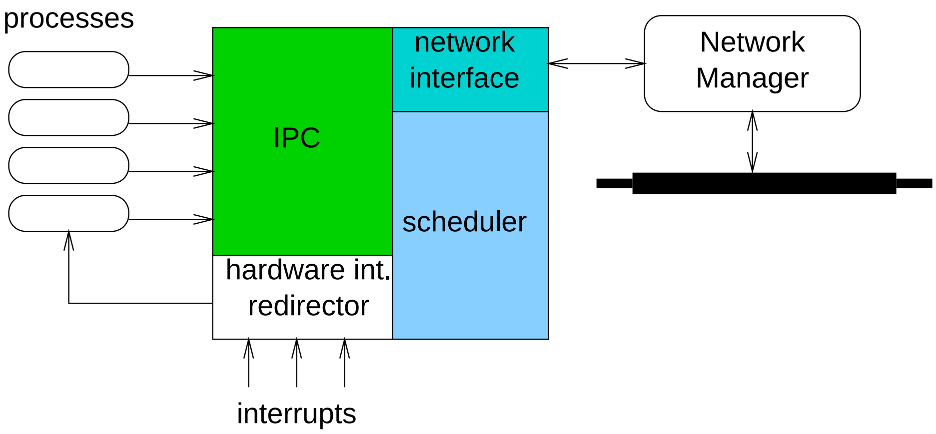
QNX integrates the entire network into a single, homogeneous set of resources. Any process on any machine in the network can directly make use of any resource on any other machine. There is no difference between a local or remote resource and no special facilities need to be built into applications to make use of remote resources. Users may access files anywhere on the network, take advantage of any peripherial device, and run applications on any machine on the network. Processes can communicate in the same manner anywhere throughtout the entire network.

Chapter 2

QNX's microkernel

The QNX microkernel is responsible for the following:

- **Interprocess communication**. The Kernel supervises the routing of *messages (cf. Section 3)*; it also manages two other forms of IPC: *proxies (cf. Section 4)* and *signals (cf. Section 5)*.
- **Low-level network communication (cf. Section 6)**. The Kernelde-livers all messages destined for processes on the other nodes.
- **Process scheduling (cf. Section 7)**. The Kernel's *scheduler* decides which process will execute next.
- **First-level interrupt handling**. All hardware interrupts and faults are first routed through the Kernel, then passed on to the appropriate driver or system manager.


processes
IPC
network
interface
scheduler
hardware int.
redirector
interrupts
Network
Manager

Chapter 3

IPC via messages

In QNX a message is a packet of bytes that is synchronously transmitted from one process to another. QNX attaches no meaning to content of a message. To communicate directly with one another, cooperating processes use these C language functions:

- *Send()*[1] for sending messages,
- *Receive()*[2] for receiving messages, and
- *Reply()*[3] for replying to process that have sent messages.

These functions can be used locally or across the network.

Next picture illustrates outlines a simple sequence of events in which two processes, Process A and Process B, use *Send()*[4], *Receive()*[5], and *Reply()*[6]to communicate with each other:

[1] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/send.html

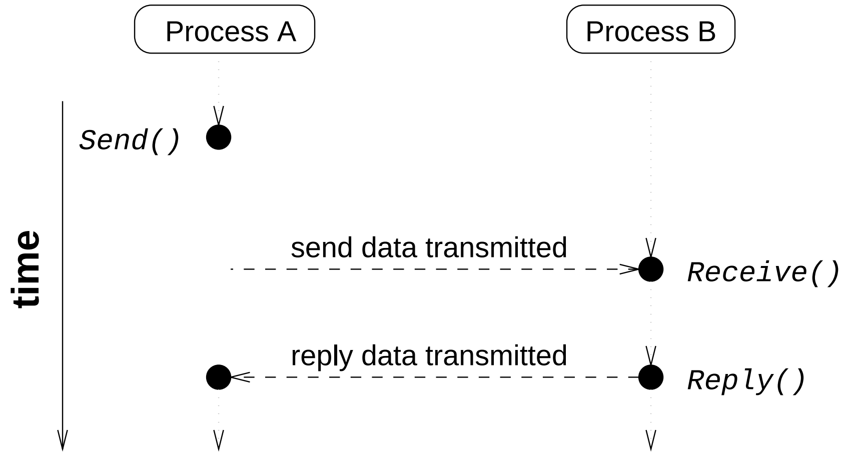
[2] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/receive.html

[3] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/reply.html

[4] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/send.html

[5] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/receive.html

[6] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/reply.html



1. Process A sends a message to Process B by issuing a *Send()*[7] request to the Kernel. At this point, Process A becomes SEND-blocked until Process B issues a *Receive()*[8] to receive the message.

2. Prtocess B issues a *Receive()*[9] and receives Process A's waiting message. Process A changes to a REPLY-blocked state. Since a massege was waiting, Process B does not block.
   If Process B has issued the *Receive()*[10] before a message was sent, it would become RECEIVE-blocked until a message arrived. In this case, the sender would immediately go into the REPLY-blocked state when it sent its message.

3. Propcess B completes the processing associated with the message it received from Process A and issues a *Reply()*[11]. The reply message is copied to Process A, which is made ready to run. A runs depends on the relative priorities of Process A and Process B.

[7] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/send.html

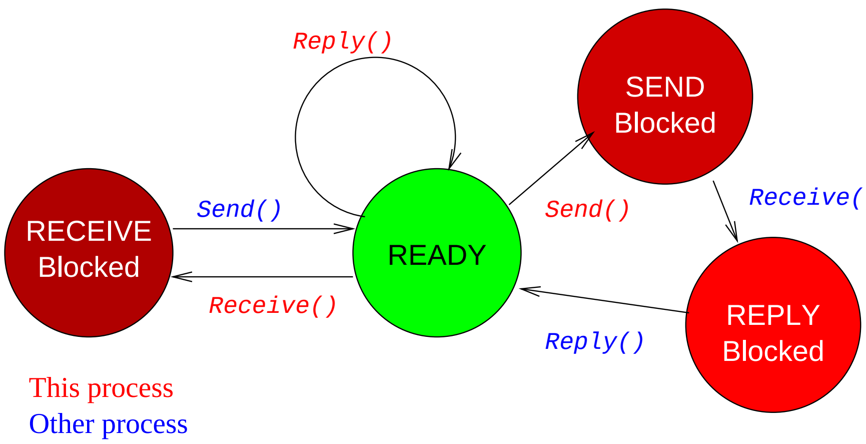
[8] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/receive.html

[9] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/receive.html

[10] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/receive.html

[11] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/reply.html

3.1 Blocked states



When a process is not allowed to continue executing, because it must wait for some part of the message protocol to end, the process is said to be *blocked*. The following table summarizes the blocked states of processes:

If a process has issued a:	**the process is:**
Send()[12] request, and the message it has sent has not yet received by the recipient process	SEND-blocked
Send()[13] request, and the message has been received by the recipient process, but that process has not yet replied	REPLY-blocked
Receive()[14] request, but has not yet received a message	RECEIVE-blocked

3.2 Advanced facilities

The QNX provides these advanced massage-passing facilities:

- **Conditional message reception**

  If a process need determine whether messages are pending without becoming RECEIVE-blocked, the process could use the *Creceive()*[15] function to read a message, if one became available, yet return immediately if no further messages were pending.

[15] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/creceive.html

Creceive()[16] should be avoided if it is possible, because it allows to consume the processor continuously at its priority level.

- **Reading or writing part of a message**

  Sometimes it is desirable to read or write only a part of a message at a time so that you can use the buffer space already allocated for the message instead of allocating a separate work buffer. It can be done by using *Readmsg()*[17] and *Writemsg()*[18] C-functions.

- **Multi-part messages**

  A message may have a fixed length header followd by a variable amount of data. So that it components will be efficiently sent or received without being copied into a temporary work buffer, a multi-part message can be constructed from two or more separate message buffers. This facility helps QNX I/O managers, such as **Dev** and **Fsys**, achive their high performance. The following functions are avaliable to handle multi-part messages:

  - *Creceivemx()*[19]
  - *Readmsgmx()*[20]
  - *Receivemx()*[21]
  - *Sendmx()*[22]
  - *Writemsgmx()*[23]

  Multi-part messages can be specified with az mx control structure. The kernel assembles these into a single data stream:

[16] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/creceive.html

[17] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/readmsg.html

[18] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/writemsg.html

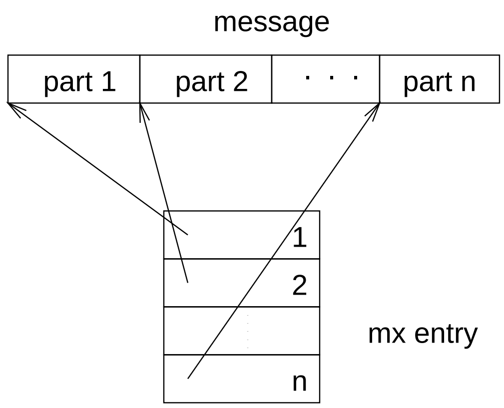
[19] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/crecvmx.html

[20] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/readmsgm.html

[21] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/receivem.html

[22] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/sendmx.html

[23] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/writemsx.html



3.3 Reserved message codes

Quantum begins all of its messages with a 16-bit word called *message code*. Note that Quantum's system processes use message codes in the following ranges:

`0x0000` to `0x00FF`	Process Manager messages
`0x0100` to `0x01FF`	I/O messages (common to all I/O services)
`0x0200` to `0x02FF`	Filesystem Manager messages
`0x0300` to `0x03FF`	Device Manager messages
`0x0400` to `0x04FF`	Network Manager messages
`0x0500` to `0x0FFF`	Reserved for future Quantum system processes

Chapter 4

IPC via proxies

A *proxy* is a form of non-blocking message especially suited for event notification where the sending process does not need to interact with the recipient. The only function of a proxy is to send a fixed message to a specific process that owns the proxy. Like messages, proxies work across the network. By using a proxy, a process or an interrupt handler can send a message to another process without blocking or having to wait for a reply.

Proxies are created with the *qnx_proxy_attach()*[1] C function. Any other process or any interrupt handler that knows the identification of the proxy can then cause the proxy to deliver its predefined message by using the *Trigger()*[2] C function. The Kernel handles the *Trigger()*[3] request. A proxy can be triggered more than once -it sends a message for each time it's triggered. A proxy process can queue up to 65,535 messages for delivery.

[1] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/qprox.at.html

[2] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/trigger.html

[3] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/trigger.html

Chapter 5

IPC via signals

QNX supports a rich set of POSIX signals, in addition to QNX-specific signals as well as signals historically found on some UNIX systems. If you want to generate a signal from the shell you can use **kill** or **slay** utilities, within a process *kill()*[1] or *raise()*[2] C functions can be used. A process can receive a signal in one of three different ways, depending on how it has dfined its signal-handling environment:

1. If the process has taken no special action to hanlde signals, the default action for the signal is taken. Usually, this action is to terminate the process.

2. The process can ignore the signal. If a process ignores a signal, there is no effect on the process when the signal is delivered. Some signals, such as SIGCONT, SIGKILL, and SIGSTOP can not be ignored under normal cicumstances.

3. The process can provide a *signal handler* for the signal. This is a function in the process that is invoked when the signal is delivered. When a process contains a signal handler for a signal, it is able to catch the signal. No data is transferred with the signal.

To define the type of handling you want for each signal, you use the ANSI C *signal()*[3] function or the POSIX *sigaction()*[4] function. Only some POSIX

[1] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/kill.html
[2] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/raise.html
[3] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/signal.html
[4] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/sigact.html

and ANSI C library functions are specified as being safe to use within signal handlers. You should not try to use any other library functions, since the results of doing so are unspecified and unpredictable. Nor should you try to use any user functions in your program unless they are re-entrant.

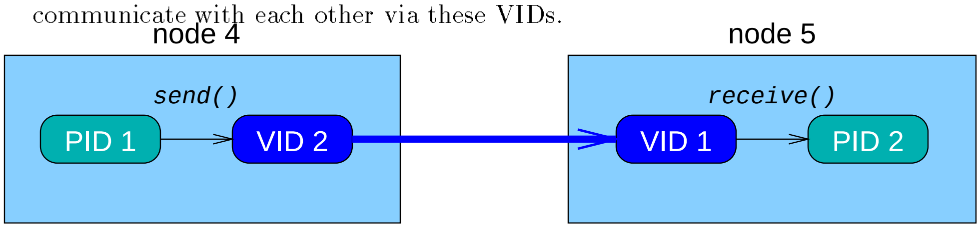
Chapter 6

IPC across the network

A QNX application can talk to a process on another computer on the network just as if it were talking to another process on the same machine. From the application's perpective, there is no difference between a local and remote resource. This transparency is made possible by *virtual sircuits* (VCs), which are paths the Network Manager provides to transmit messages, proxis, and signals across the network. VCs contribute to efficient overall use of resources in a QNX network for several reasons:

- When a VC is created, it has given the ability to handle messages up to a specified size; this means you can preallocate resources to handle the message. When you need to send a message larger than the maximum specified size, the VC is automatically resized to accomodate the larger message.
- If two process residing on different nodes are communicating with each other via more than one VC, the VCs are shared, only one *real* virtual circuit exists between the processes. This situation occurs commonly when a process acesses several files on the remote filesystem.
- If a process attaches to an existing shared VC and it requests a bufefr size larger than that currnetly in use, the buffer size is automatically increased.
- When a process terminates, its associated VCs are automatically released.

A sending process is responsible for setting up the VC between itself and the process it wants to communicate with. To do so the sending process usually issues a *qnx_vc_attach()*[1] function call. in addition to creating a VC, this call also creates a virtual process ID, or VID, at each end of the circuit. To the process at either end of the virtual circuit, the VID on its end appears to have the process ID of the remote process it wants to communicate with. Processes communicate with each other via these VIDs.



In the previous example the virtual circuit connects PID 1 to PID 2. On node 4 where PID 1 resides a VID represents PID 2. On node 5 where PID 2 resides, a VID represents PID 1. Both PID 1 and PID 2 can refer to the VID on their node as if it were any other local process, sending messages, receiving messages, raising signals, waiting, etc. For example, PID 1 can send a message to the VID on its end, and this VID will relay the message across the network to the VID representing PID 1 on the other end. This VID will then route the message to PID 2. Each VID maintains a connection that contains the following information:

- local *pid*
- remote *pid*
- remote *nid* (node ID)
- remote *vid*

You do not need to contact VC directly. When az application wants to access an I/O resource across the network, a VC is created by the *open()* function. To the application, the VC simply appears to be a PID.

A process might become unable to communicate over an established VC for various reason:

- The computer it was running on was powered down.
- The network cable to the computer was disconnected.
- The remote process it was communicating with was terminated.

[1] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/qvc.at.html

The process manager on each node checks the integrity of VCs on its node. It does this as follows:

1. Each time a successful transmission takes place on a VC, a time stamp associated with the VC is updated to indicate the time of last activity.
2. At installation-defined intervals, the Process Manager looks at each VC. If there's been no activity on a circuit, the Process Manager sends a network integrity packet to the Process Manager on the node at the other end of the circuit.
3. If no response comes back, or if a problem is indicated, the VC os flagged as having a problem. An installation-defined number of attempts are then made to re-establish contact.
4. If the attempts fail, the VC is dismantled; any processes blocked on the VC is made READY. (The process sees a failure return code from the communication primitive on the VC.)

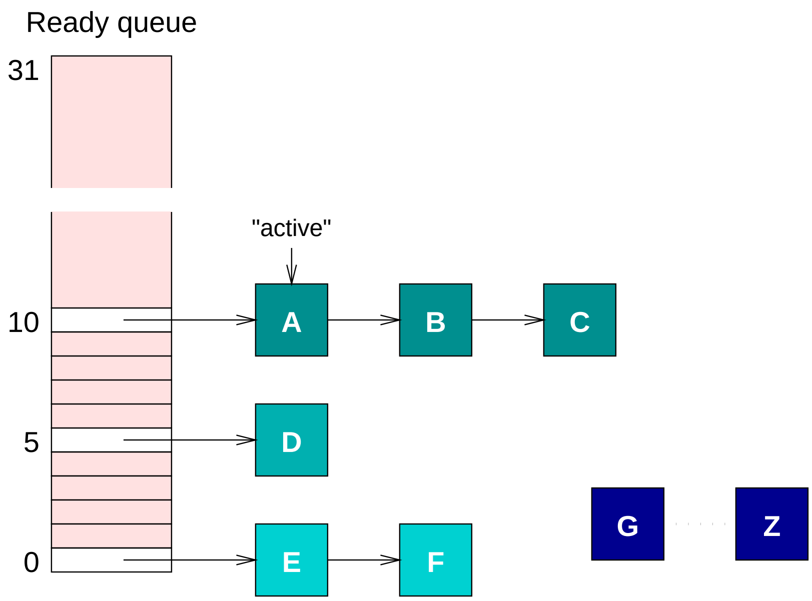
Chapter 7

Process scheduling

The Kersel's scheduler makes scheduling decisions when:

- a process becomes unblocked;
- the timeslice for a running process expires;
- a running process is preempted.

In QNX, every process has its own priority. The scheduler selects the next process to run by looking at the priority assigned to every process that is READY. The process with the highest priority is selected to run.



The prioriries assigned to processes range from 0 (the lowest) to 31 (the highest). The default priority for a new process is inherited from its parent, this is normally set to 10 for programs started by the shell. You can use *getprio()*[1] C function to determine the priority of a process and *setprio()*[2] C function to set the priority of a process. QNX provides three scheduling methods: FIFO, round-robin, and adaptive scheduling. Each process on the system my run any one of these methods. These scheduling methods apply only when two or more processes that shere the same priority are READY. If a higher priority process becomes ready, it immediatelly preempts all lower-priority processes.

The scheduling method of a process is inherited from its parent process, but it can be changed using *sched_setscheduler()*[3] C function or it cab be determined by the *sched_getscheduler()*[4] C function.

---

[1] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/getprio.html

[2] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/setprio.html

[3] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/scsetsc.html

[4] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/scgetsc.html

7.1 FIFO scheduling

In FIFO scheduling, a process selected to run continues executing until it:

- voluntarily relinquishes (e.g. it blocks)
- is preempted by a higher-priority process

Two process that run at the same priority level can use FIFO scheduling to ensure mutual exclusion to a shared resource. Neither process will be preempted by the other while it is executing.

7.2 Round-robin scheduling

In round-robin scheduling, a process selected to run continues executing until it:

- voluntarily relinquishes control
- is preempted by a higher-priority process
- consumes its *timeslice*

A timeslice is the unit of time assigned to every process. Once it consumes its timeslice, a process is preempted and the next READY process at the same priority level is given control. A timeslice is 100 milliseconds. Apart from time slicing, round-robin scheduling is identical to FIFO scheduling.

7.3 Adaptive scheduling

In adaptive scheduling, a process behaves as follows:

- If the process consumes its timeslice (i.e. it does not block), its priority is reduced by 1 if another process at the same priority is READY. This is known as *priority decay*.
- If the process has decayed and remains unscheduled for one second, its priority is boosted by 1 (a process is never boosted above its original priority).

- If the process blocks, it immediately reverts to its original priority.

You can use adaptive scheduling in envirinments where potentially compute-intensive background processes are shering the computer with interactive users. You should find that adaptive scheduling gives retains fast interactive response for other processes. Adaptive scheduling is the default scheduling method for programs created by the shell.

Chapter 8

Process Manager

The process manager works closely with the Kernel, although it shares the same address space as the Kernel, and it is the only process to do so, the Process Manager runs as a true process. It is scheduled to run by the Kernel like all other processes and it uses the Kernel's message passing primitives to communicate with other processes.

The Process Manager is responsible for creating new processes in the system and managing the most fundamental resources associated with a process. These services are all provided via messages. You can easily create a process on another node by sending the process creation message to the Process Manager on that node.

8.1 Process creation primitives

QNX supports three process creation primitives:

fork() [1] The *fork()*[2] primitive creates a new process that is an exact image of the calling process. The new process shares the same code as the calling process and inherits a copy of all the calling process's data.

[1] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/fork.html
[2] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/fork.html

exec() [3] The *exec()*[4] primitive replaces the calling process image with a new process image. There is no return from a successful *exec()*[5], because the calling process image is overlaid by the new process image. It is common practice in POSIX systems to create a new process by first calling *fork()*[6], and then having the child of the *fork()*[7] call *exec()*[8].

spawn() [9] The *spawn()*[10] primitive creates a new process as a child of the calling process. It can avoid the need to *fork()*[11] and *exec()*[12], resulting in a faster and more efficient means for creating new processes. Unlike *fork()*[13] and *exec()*[14], which by very nature operate on the same node as the calling process, the *spawn()*[15] primitive can create processes *on any node in the network.*

The *spawn()*[16] is unique to QNX while *fork()*[17] and *exec()*[18] are defined by POSIX.

8.2 Process inheritance

When a process is created by one of the three primitives described above, it inherits much of its environment from its parent:

[3] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/exec.html
[4] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/exec.html
[5] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/exec.html
[6] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/fork.html
[7] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/fork.html
[8] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/exec.html
[9] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/spawn.html
[10] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/spawn.html
[11] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/fork.html
[12] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/exec.html
[13] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/fork.html
[14] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/exec.html
[15] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/spawn.html
[16] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/spawn.html
[17] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/fork.html
[18] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/exec.html

Item inherited	*fork()*[19]	*exec()*[20]	*spawn()*[21]
process ID	no	yes	no
open files	yes	optional	optional
file locks	no	yes	no
pending signals	no	yes	no
signal mask	yes	optional	optional
ignored signals	yes	optional	optional
signal handlers	yes	no	no
environment variables	yes	optional	optional
session ID	yes	yes	optional
process group	yes	yes	optional
real UID, GID	yes	yes	yes
effective UID, GID	yes	optional	optional
current working directory	yes	optional	optional
file creation mask	yes	yes	yes
priority	yes	optional	optional
scheduler policy	yes	optional	optional
virtual circuits	no	no	no
symbolic names	no	no	no
real-time timers	no	no	no

The loading of process images is done by a *loader thread*. The loader code resides in the Process Manager, but the thread runs under the process ID of the new process. Once a program code has been loaded, the process is ready for execution. Note that all processes run concurrently with their parents. In addition, the death of a parent process does *not* automatically cause the death of its child process.

8.3 Process termination

A process is terminated in either of two ways:

1. a signal whose defined action is to cause process termination is delivered to the process;
2. the process invokes the *exit()*[22] C function, either explicitly or by default action when returning from *main()*.

Termination involves two stages:

[22] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/exit.html

1. A *termination thread* in the Process Manager is run. This "trusted" code is in the Process manager but the thread runs with the process ID of the terminating process. This thread closes all open file descriptors and releases the following:
    - any virtual circuits held by the process
    - all memory allocated to the process
    - any symbolic names
    - any major device numbers (I/O managers only)
    - any interrupt handlers
    - any proxies
    - any timers
2. After a termination thread is run, notification of process termination is sent to the parent process (this phase runs inside the Process Manager).

    If the parent process has not issued a *wait()*[23] or *waitpid()*[24] call, the child process becomes a "zombie" process and will not terminate until the parent process issues a *wait()*[25] or terminates. If you do not want a process to *wait* on the death of children, you should either set the _SPAWN_NOZOMBIE flag with *qnx_spawn()*[26] or *qnx_spawn_options()*[27] or set the action for SIGCHLD to be SIG_IGN via *signal()*[28]. This way, children will not become zombies when they die.

    A parent process can *wait* on the death of a child spawned on a remote node. If the parent of a zombie process terminates, the zombie is released.

If a process is terminated by the delivery of a termination signal and the **dumper** utility is running, a memory image dump is generated. You can examine this dump with the symbolic debugger.

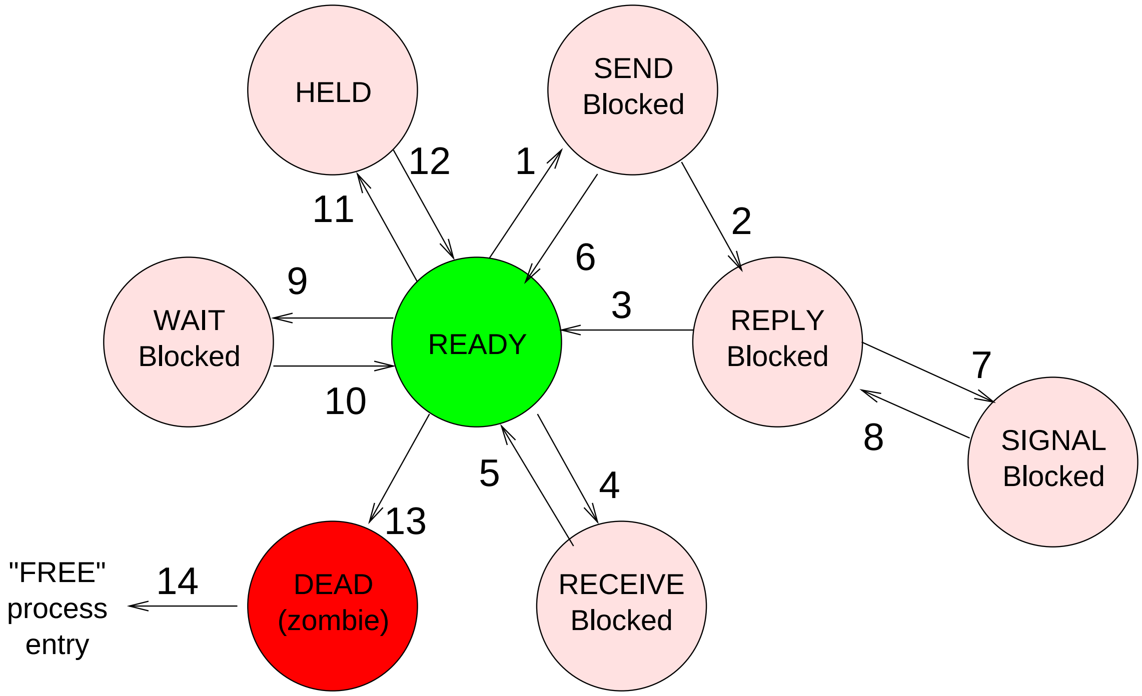
8.4 Process states

A process is always in one of the following states:

[23] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/wait.html
[24] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/waitpid.html
[25] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/wait.html
[26] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/qspawn.html
[27] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/qspawn.o.html
[28] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/signal.html

- **READY**. The process is capable of useing the CPU (i.e. it is not waiting for any event to occur).
- **BLOCKED**. The process is in one of the following blocked states:
  - SEND-blocked
  - RECEIVE-blocked
  - REPLY-blocked
  - SIGNAL-blocked
- **HELD**. The process has received a SIGSTOP signal. Until it is removed from the HELD state, a process is not eligible to use the CPY; the only way to remove it from the HELD state is to either deliver a SIGCONT signal or terminate the process via a signal.
- **WAIT-blocked**. The process has issued a *wait()*[29] or *waitpid()*[30] call to wait for status from one or more of its child processes.
- **DEAD**. The process has terminated but is unable to send its exit status to its parent beause the parent has not issued a *wait()*[31] or *waitpid()*[32]. A DEAD process has a state, but the memory it once occupied has been released. A DEAD process is also known as a *zombie* process.

[29] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/wait.html
[30] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/waitpid.html
[31] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/wait.html
[32] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/waitpid.html



The transactions are follows:

1. Process sends message.
2. Target process receives message.
3. Target process replies.
4. Process waits for message.
5. Process receives message.
6. Signal unblocks process.
7. Signal attempts to unblock process;
   target process has requested message signal catching.
8. Target process receives signal message.
9. Process waits on death of child.
10. Child dies or signal unblocks process.
11. SIGSTOP set on process.

12. SIGCONT set on process.
13. Process dies.
14. Parent waits on death, terminates itself or has already terminated.

To determine the state of an individual process from within the shell, you can use **ps** and **sin** utilities; from within applications, use *qnx_psinfo()*[33] function. To determine the state of the operating system as a whole, use **sin** utility from the shell, and *qnx_osinfo()*[34] function from applications.

8.5 Process symbolic names

QNX encourages the development of applications that are split up into cooperating processes. An application that exists as a team of cooperating processes exhibits greater concurrency and can be network-distributed for greater performance.

Splitting up applications into cooperating processes requires special considerations, however. If cooperating processes are to communicate reliably with each other, they must be able to determine each other's process ID. For example, let's say you have a database server process that provides services to an arbitrary number of clients. The clients start and stop at any time, but the server always remains available. How do client processes discover the process ID of the database server so they can send messages to it?

In QNX, the solution is to let processes give themselves a symbolic name. In the context of a single node, processes can register this name with the Process Manager on the node where they are running. Other processes can then ask the Process Manager for the process ID associated with that name.

The situation becomes more complex in a network environment where a server may need to service clients from several nodes across a network. QNX accordingly provides the ability to support both global names and local names. Global names are known accross the network, whereas local names are known only on the node where they are registered. Global names start with a slash(/). In order for global names to be used, a process known as a *process name locator* (i.e. **nameloc** utility) must be running on at least one node of a network. This process maintains a record of all global names that have been registered.

[33] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/qpsinfo.html

[34] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/qosinfo.html

Up to ten process name locators may be active on a network at a particular time. Each maintains an identical copy of all active global names. This redundancy ensures that a network can continue to function properly even if one or more nodes supporting process name location fail simultaneously.

To attach a name, a *server* process uses the *qnx_name_attach()*[35] C function. To locate a process by name, a *client* process uses the *qnx_name_locate()*[36] C function.

8.6 Timers

In QNX, time management is based on a system timer maintained by the operating system. The timer contains the current Coordinated Universal Time (UTC) relative to 0 hours, 0 minutes, 0 seconds, January 1, 1970. To establish local time, time management functions uses the **TZ** environment variable.

Shell scripts and processes can pause for a specific number of seconds. For shell scripts this facility is provided by the **sleep** utility; for processes, it is provided by the *sleep()*[37] C function. The *delay()*[38] function takes a time interval specified in milliseconds.

A process can create one or more timers using *timer_create()*[39] C function. This function lets specify the type of event-reporting mechanism to be used:

- **sleep until completion**: the prcess will sleep from the time it issues the timer arming call until the timer interval expires
- **notify with proxy**: a proxy will be used to notify th eprocess that the time interval has expired
- **notify with signal**: a user specified signal will be delivered to the process when the time interval expires.

You can also have a timer go off repeatedly at a specified interval. For example let's say you have a timer armed to go off at 9 am tomorrow morning. You can specify that it should also go off every five minutes thereafter.

[35] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/qname.at.html

[36] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/qname.lo.html

[37] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/sleep.html

[38] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/delay.html

[39] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/timercre.html

To arm timers with an absolute or relative time interval, use *timer_settime()*[40] C function.

To remove timers, you can use the *timer_delete()*[41] C function. A timer can remove itsef when its time interval expires, provided these conditions are met:

- the TIMER_AUTO_RELEASE option was included in the *timer_settime()*[42] function call
- the timer was not armed to go off repeatedly.

Resolution of a timer can be set via the **ticksize** utility or the *qnx_ticksize()*[43] C function. To inspect the interval outstanding on a timer, or check if the timer has benn removed, you use the *timer_gettime()*[44] C function.

[40] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/timerset.html

[41] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/timerdel.html

[42] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/timerset.html

[43] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/qticksize.html

[44] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/timerget.html

Chapter 9

I/O Namespace

This document is not ready yet:-(

Chapter 10

Filesystem Manager

In QNX a *file* is an object that can be written to, read from, or both. QNX implements six types of files; five of these are managed by **Fsys**:

- **Regular files** consist of randomly accessible sequences of bytes and have no other predefined structure.
- **Directories** contain the information needed to locate regular files; they also contain status and attribute informationfor each regular file.
- **Symbolic links** contain pathname to a file or directory that is to be accessed in place of the symbolic link file; these files are often used to provide multiple paths to a single file.
- **Pipes** and **FIFOs** serve as I/O channels between cooperating processes.
- **Block special files** refer to devices, such as disk drives, tapes, and disk drive partitions; these files are normally accessed in a manner that hides the hardware characteristics of the device from applications.

Fsys maintains four different times for each file:

1. date of last access (read)
2. date of last write
3. date of last modification
4. date of creation

10.1 Regular files and directories

10.1.1 Regular files

QNX views a *regular file* as a randomly accessible sequence of bytes that has no other predefined internal structure. Applications are responsible for understanding the structure and content of any regular file.

10.1.2 Directories

A *directory* is a file that contains *directory entries*. Each directory entry associates a *filename* with a file. A filename is a symbolic name that lets you identify and access a file. A file may be identified by more than one filename. Although a directory behaves much like a standard file, the Filesystem Manager imposes some restrictions on the operations you can perform on a directory. You can not open a directory for writing, nor can you link to a directory with the *link()*[1] C function.

To read directory entries, you can use a set of POSIX-defined C finctions. These functions include:

- *opendir()*[2]
- *readdir()*[3]
- *rewinddir()*[4]
- *closedir()*[5]

Since QNX directories are simply files that contaion "known" information, you can also read directory entries directly using the *open()*[6] and *read()*[7] functions. This technique is not portable, however, the format of directory entries varies from operating system to operating system.

---

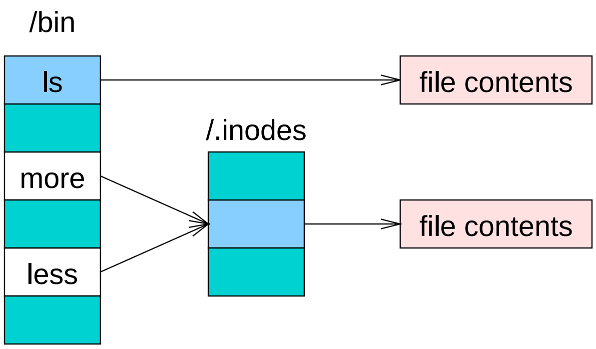
[1] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/link.html
[2] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/opendir.html
[3] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/readdir.html
[4] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/rewinddi.html
[5] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/closedir.html
[6] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/open.html
[7] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/read.html

10.2 Links and inodes

In QNX, file data can be referenced by more than one name. Each filename is called a *link*. There are actually two kinds of links: hard links, which we refer to simply as "links", and symbolic links. To support links for each file, the filename information is separated from the other information that describes a file. The non-filename information is stored in a structure called an *inode*. Note that you can create a link to a file only if the file and the link are in the same filesystem.

If a file has only one link, the inode information is stored in the directory entry for the file, otherwise the inode is stored as a record in a special file named **/.inode**, as are the file's directory entry points to the inode record. There are two other cases in which a file can have an entry in the **/.inodes** file:

- If a file's name is longer than 16 characters, the inode information is stored in the **/.inode** file, making room for a 48 character filename in the directory entry.
- If a file has had more than one link and all links but one have been removed, the file continues to have a separate **/.inodes** file entry.



You can create links from within the shell using **ln** utility, or from within programs with *link()*[8] C function. You can not create hard links to directories.

When a file is created, it is given a *link count* of one. As links to the file are added, this link count is incremented; as links to the file are removed, the link count is decremented. The file is removed when its link count goes to zero and all programs using the file have closed it. To remove links from whitin the shell

[8] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/link.html

use **rm** utility, or from within programs use *remove()*[9] or *unlink()*[10] C functions.

10.3 Symbolic links

A *symbolic link* is a special file that has a pathname as its data. When a symbolic link is named in an I/O request (by *open()*[11], for example), the link portion of the pathname is replaced by the link's "data" and the path is re-evaluated. Unlike hard links, symnolic links can cross filesystems and can also create links to directories.

Several functions operate directly on the symbolic link. For these functions, the replacement of the symbolic element of the pathname with its target is not performed. These functions include *unlink()*[12] (which removes the symbolic link), *lstat()*[13], and *readlink()*[14].

10.4 Pipes and FIFOs

10.4.1 Pipes

A *pipe* is an unnamed file that serves as an I/O channel between two or more cooperating processes, one process writes into the pipe, the other reads from the pipe. The Filesystem Manager takes care of buffering the data. A typical application for a pipe is connectiong the output of one program to the input of another program. This connection is often made by the Shell. For example:

ls | more

directs the standard output from the **ls** utility through a pipe to the standard input of the **more** utility. If you want to create pipes from within the shell, use the pipe symbol ("|"), or if you want to create pipes from within programs, use *pipe()*[15] or *popen()*[16] C functions.

[9] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/remove.html
[10] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/unlink.html
[11] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/open.html
[12] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/unlink.html
[13] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/lstat.html
[14] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/readlink.html
[15] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/pipe.html
[16] See URL

On diskless workstations, you can run the Pipe Manager (**Pipe**) inplace of the Filesystem Manager when only pipes are required. The Pipe Manager is optimized for pipe I/O.

10.4.2 FIFOs

FIFOs are essentially the same as pipes, except that FIFOs are named permanent files that are stored in filesystem directories. FIFOs can be created by **mkfifo** utility from within the shell, or *mkfifo()*[17] C function from within programs. To remove FIFOs, use **rm** utility or *remove()*[18] or *unlink()*[19] function.

---

http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/popen.html

[17] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/mkfifo.html

[18] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/remove.html

[19] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/unlink.html

Chapter 11

Device Manager

The QNX Device Manager (**Dev**) is the interface between processes and *terminal devices*. These terminal devices are located in the I/O namespace with names starting with **/dev**. For example, a console device on QNX would have a name such as: **/dev/con1**.

QNX programs access terminal devices using the standard *read()*[1], *write()*[2], *open()*[3], and *close()*[4] functions. A terminal device is presented to a QNX process as a bidirectional stream of bytes that can be read or written by the process. The Device Manager regulates the flow of data between an application and the device. Some processing of this data is performed by **Dev** according to parameters in a *terminal control structure* (called *termios*), which exists for each device. Users can query and/or change parameters by using the **stty** utility; programs can use the *tcgetattr()*[5] and *tcsetattr()*[6] functions. The *termios* parameters control low-level fuctionality such as:

- line-control discipline (including baud rate, parity, stopbits, and data bits)
- echoing of characters
- input line editing

[1] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/read.html
[2] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/write.html
[3] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/open.html
[4] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/src/close.html
[5] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/tcgetatr.html
[6] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/tcsetatr.html

- recognizing, and acting on, breaks and hangups
- software and hardware flow control
- translation of output characters

Some auxiliary services provided by the Device Manager:

- *dev_read()*[7] or *read()*[8] + *tcsetattr()*[9]: perform timed read operations.
- *dev_arm()*[10]: asynchronously notify a process of data available on one or more input devices.
- *tcdrain()*[11]: wait for output to be completely transmitted.
- *tcsendbreak()*[12]: send breaks across a communications channel.
- *tcdropline()*[13]: disconnect a communications channel.
- *dev_insert_chars()*[14]: insert input data.
- *open()*[15] and *fcntl()*[16] (O_NONBLOCK mode): perform non-blocking reads and writes.

11.1 Edited input mode

The most significant mode ofdevice processing is controlled by the ICANON bit in the *termios* control structure. If this control bit is set, the Device Manager performs line-editing functions on received characters. Thus, only when line is

[7] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/dev.read.html

[8] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/read.html

[9] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/tcsetatr.html

[10] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/dev.arm.html

[11] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/tcdrain.html

[12] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/tcsndbrk.html

[13] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/tcdrplin.html

[14] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/dev.ins.html

[15] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/open.html

[16] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/fcntl.html

entered (typically when a carriage return is received) will the processed data be made available to applications. This mode of operation is referred to as *edited*, *canonical*, or *cooked* mode. Line editing characters vary from terminal to terminal. The QNX console always starts out with a full set of editing keys defined.

If a terminal is connected to QNX via a serial channel, it is necessary to define the editing characters that apply to that particular terminal. T do this, you can use **stty** utility.

11.2 Raw input mode

When ICANON is not set, the device is said to be in *raw* mode. In this mode, no input editing is performed, and any received data is made immediately available to QNX processes. When reading from a *raw* device, the application is capable of specifying underwhat conditions an input is to be satisfied. The criteria used for accepting raw input data are based on two members of the *termios* control structure: MIN and TIME. The application can specify a further qualifier of accepting input data when it issues its read request via *dev_read()*[17]. This qualifier, TIMEOUT, is useful when writing protocols or realtime applications. TIMEOUT is always 0 for *read()*[18]. When a QNX process issues a read request for n bytes of data, these three paraeters define when that read request is to be satisfied:

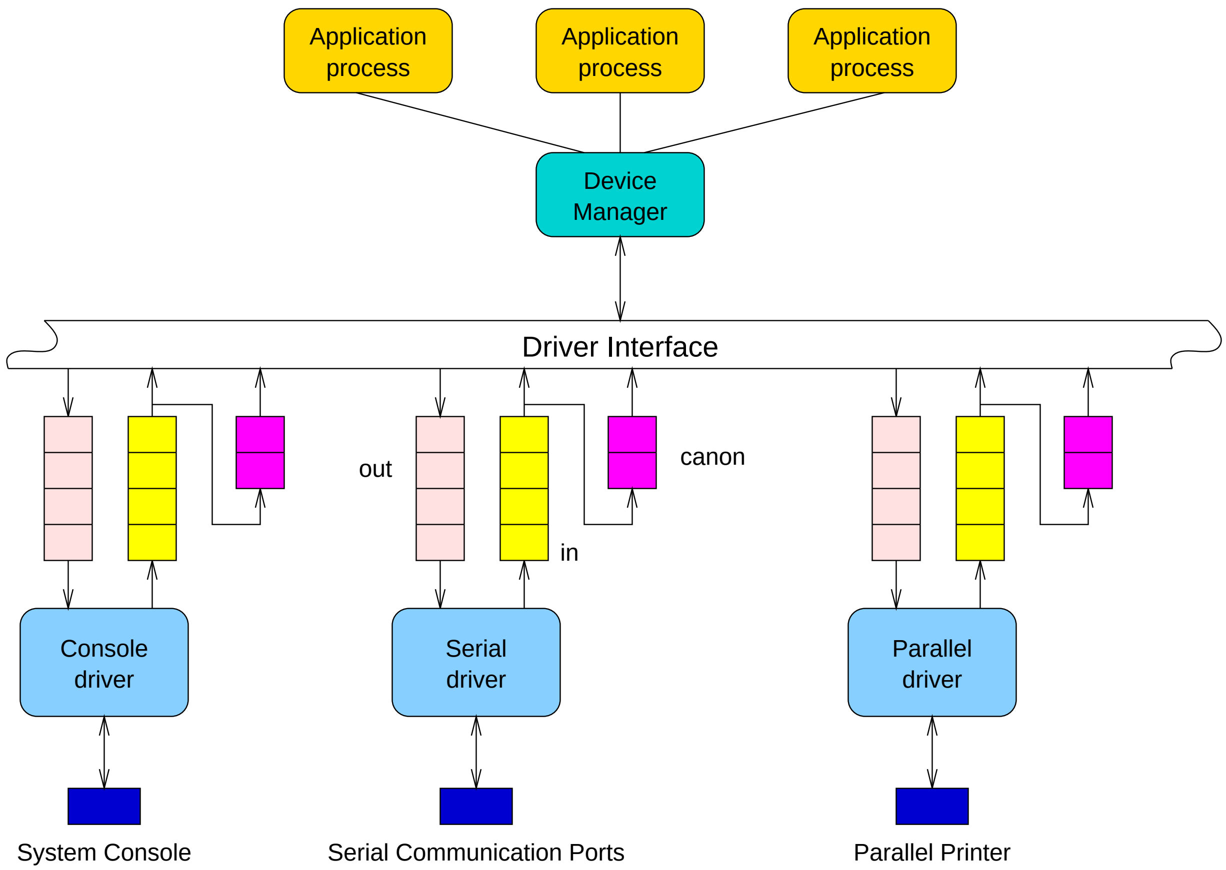
[17] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/dev.read.html

[18] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/read.html

MIN	TIME	TIMEOUT	Description
0	0	0	Return immediately with as many bytes as are currently available (up to *n* bytes).
M	0	0	Return with up to *n* bytes only, when at least **M** bytes are available.
0	T	0	Return with up to *n* bytes when at least one byte is available, or **T**x0.1 second has expired.
M	T	0	Return with up to *n* bytes when either **M** bytes are available or at least one byte has been received and the interbyte time between any subsequently received characters exceeds **T**x0.1 second.
0	0	t	Reserved.
M	0	t	Return with up to *n* bytes when **t**x0.1 second has expired, or **M** bytes are available.
0	T	t	Reserved.
M	T	t	Return with up to *n* bytes when **M** bytes are available, or **t**x0.1 second has expired and no characters are received, or at least one byte has been received and the interbyte time between any subsequently received characters exceeds **T**x0.1 second.

11.3 Device drivers

A Typical QNX device subsystem:



The hardware interface is managed by individual *driver* processes. Data flows between **Dev** and its *drivers* through a set of shared memory queues for each *terminal device*. Because shared memory queues are used, it is necessary that **Dev** and all of its drivers reside on the same physical CPU. Three queues are used for each device. Each queue is implemented using a first-in, first-out mechanism. A control structure is also associated with each queue.

Received data is placed into the raw input queue by the driver and is consumed by **Dev** only when application process request data. Interrupt handlers within drivers typically call a trusted library routine within **Dev** to add data to this queue. **Dev** places output data into the output queue; the data is consumed by the driver as characters are physically transmitted to the device. **Dev** calls a trusted routine within the driver process each time new data is added so it can "kick" the driver into operation. Since output queues are used, **Dev** implements full *write-behind* for all *terminal devices*. Only when the output buffers are full will **Dev** cause a process to block while writing.

The canonical queue is managed by **Dev** and is used while processing input data in *edited* mode. The size of this queue determines the maximum edited input line that can be processes for a particular device. The sizes of all these queues are configurable; the only restriction is that the sum of all three queues can not exceed 64K.

Device drivers simply add received data to the raw input queue or consume and transmit data from the output queue. **Dev** decides when and if output transmission is to be suspended, how and if received data is echoed, etc. Low-level device control is implemented with a far call into an *ioctl* entry point within each driver. A common set of *ioctl* commands are supported by most drivers used directly by **Dev**. Device-specific *ioctl* commands can also be sent through **Dev** to the drivers by QNX processes (via the *qnx_ioctl()*[19] C function).

11.4 The QNX console

System consoles are managed by the **Dev.con** driver process. The display adapter and the screen, plus the system keyboard, are collectively referred to as the *console*. The console driver provides a set of cosole-specific functions that let application processes communicate via messages directly to a console driver process. Communication is established with the *console_open()*[20] C function. Once a communication is established, a QNX process can

- read directly from the console screen with *console_read()*[21];
- write directly to the console screen with *console_write()*[22];
- be asynchronously notified of significant events with *console_arm()*[23];
- control the console size with *console_size*[24];
- switch the visible console with *console_active()*[25].

[19] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/qioctl.html

[20] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/con.open.html

[21] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/con.rd.html

[22] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/con.wr.html

[23] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/con.arm.html

[24] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/con.siz.html

[25] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/con.actv.html

11.5 Serial devices

Serial communication channels are managed by the **Dev.ser** driver process. This driver can manage more than one physical channel; it provides terminal devices with names such as **/dev/ser1**, **/dev/ser2**, etc. **Dev.ser** is an example of a purely interrupt-driven I/O server. After initializing the hardware, the process itself goes to slepp. Received interrupts place input data directly into the input queue. The first output character on an idle channel is transmitted to the hardware when **Dev** issues the first kick call into the driver. Subsequent characters are transmitted by the appropriate interrupt being received.

11.6 Parallel devices

Parallel printer ports are managed by the **Dev.par** driver process. When you start **Dev.par**, you specify a command-line argument that determines which parallel port is installed. **Dev.par** is an output-only driver, so it has no input or canonical input queues. **Dev.par** is a typical example of a completely non-interrupt I/O server. The parallel printer process normally remains RECEIVE-blocked, waiting for data to appear in its output queue and a kick from **Dev**. When data is available to print, **Dev.par** runs in a busy-wait loop (at relatively low adaptive priority), while waiting for the printer hardware to accept characters.

Chapter 12

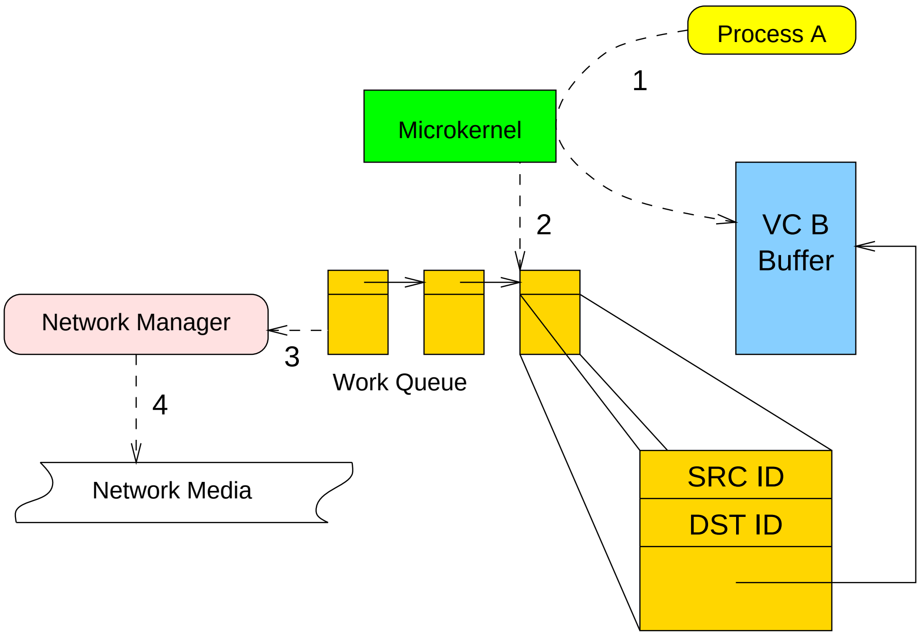
Network Manager

The Network Manager is responsible for propagating the QNX messaging primitives across a local area network. The standard messaging primitives used in local messaging are used *unmodified* in remote messaging. The Network Manager does not have to be built into the operating system image. It may be started and stopped at any time to provide or remove network messaging capabilities.

When the Network Manager starts, it registers with the Process Manager and Kernel. This activates existing code within the two that interfaces to the Network Manager. The Kernel and Process Manager interface to the Network Manager via a special nonblocking memory queue. This queue is essentially a list of transmissions to be performed by the Network Manager. The entries contain all the information for a particular operation (e.g. *Send()*[1], *Reply()*[2], VC creation, remote signal propagation, etc.).

In the case of a *Send()*[3] or *Reply()*[4] to a remote node, the following events occur:

[1] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/send.html
[2] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/reply.html
[3] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/send.html
[4] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/reply.html



1. Te process calls *Send()*[5] or *Reply()*[6] and the Kernel copies the data from the process's data space into the virtual circuit buffer associated with the VC specified by *Send()*[7] or *Reply()*[8].

2. The Kernel enqueues to the Network Manager a time-ordered queue entry identifying the sender, the remote receiver, and pointers to the data in the virtual circuit buffer. If the queue was previously empty, the Network Manager's proxy is triggered to let it know new work has arrived.

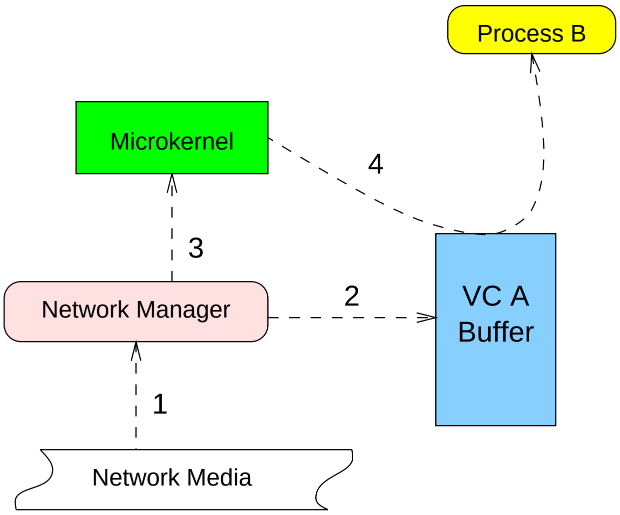
3. The Network Manager dequeues the queue entry.

4. The Network Manager begins transmission on the network media. The manager is reponsible for delivery.

In the case of a signal propagation or VC creation, the Process Manager rather than the Kernel would enqueue a control packet. Nevertheless, the Network Manager would transmit the packat to its destination.

The following events occur on the node receiving the message:

[5] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/send.html
[6] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/reply.html
[7] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/send.html
[8] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/reply.html



1. Network data arrives on the media.
2. The Network Manager copies the message data into the appropriate virtual circuit buffer.
3. The Network Manager uses a private kernel call to inform the Kernel that the reception is complete.
4. The Kernel copies the data from the virtual circuit buffer into the process's buffer (assuming it was RECEIVE- or REPLY-blocked on this virtual circuit).

Any control packets the Network Manager receives are forwarded immediately to the Process Manager via the standard *Send()*[9] primitive. These control packets are used for signal propagation and VC creation.

12.1 Network drivers

Like the Filesystem Manager and the Device Manager, the Network Manager contains no hardware-specific code. This functionality is provided by network card drivers. The Network Manager can support multiple network drivers at one time. Each driver typically supports a single network card. Shared memory queues provide the interface between the Network Manager and drivers. The

[9] See URL http://lili.iit.uni-miskolc.hu/stuff/doc/watcom/watcom/clibref/qnx/send.html

driver is responsible for packetization, sequencing, and retransmission if reliable guaranteed data transmission to a remote physical node is requested.