QNX Resource Manager

The Resource Manager is arguably the most commonly used tool in QNX programming. If you haven't written several resource managers, you might feel embarrassed to claim that you've done QNX programming. Due to its importance, QNX also has a wealth of official documentation explaining it. This article provides a detailed explanation of the basic design of QNX resource managers, the manager framework, supplemented by code. It is hoped that it will help readers master the resource managers on QNX, making it easier to build systems. The complete code for the resource manager example mentioned in the article can be found at the following link:
https://github.com/xtang2010/articles/tree/master/QNX_Resource_Manager

What is a Resource Manager?

A resource manager, as the name suggests, is a server that manages "resources." The question here is, what exactly are "resources"? In QNX, a "resource" could be a piece of hardware (a hardware resource manager is essentially what we commonly refer to as a hardware driver), a service such as TCPIP network service or NTFS file system service, or even a file (or directory).
If you remember the basic idea of Unix, which is "to treat drivers as files," then the resource manager is incredibly useful. For instance, /dev/ser1 is a resource manager for serial ports, and /dev/random is a resource manager that provides random numbers. Even traditional Unix mount points, such as the root directory / or user directory /home, can be resource managers in QNX.

Now, let's focus on the practical aspect. Suppose we want to write an MD5 resource manager; what should we do? MD5 is a service that, given a string of data from the client, calculates the MD5 value and then allows the client to retrieve it.。

A Basic Resource Manager

Essentially, a resource manager is a server. If you've read the article "Understanding QNX from the API," you should easily be able to think of the most basic resource manager as follows:
(The following is sample code. The complete code for md5name that can be compiled and run on QNX can be found on GitHub)

```
#define MD5_SEND_DATA   0x00000001
#define MD5_RECV_DIGEST 0x00000002

typedef struct
   { int msgtype;
   int msglen;
} md5_msg_t;

name_attach(...);
for (;;){
   revid = MsgReceive(chid, &msg, sizeof(msg), &info);
   switch msg.msgtype {
   case MD_SEND_DATA:
       MsgRead(rcvid, ...);
        ....
       MsgReply(rcvid, ...);
       break;
```

```
    case MD_RECV_DIGEST :
        ....
        MsgReply(rcvid, …)
        break;
    }
}
```

Yes, it's a loop that continuously receives messages and processes them according to the predefined message types. How does the client obtain this service?

```
int md5_send(int fd, unsigned char *data, int len)
{
    md5_msg_t msg;
    iov_t iov[2];

    msg.msgtype = MD5_SEND_DATA;
    msg.msglen = len;
    SETIOV(&iov[0], &msg, sizeof(msg));
    SETIOV(&iov[1], data, len);
    return MsgSendv(fd, iov, 2, 0, 0);
}

int md5_recv(int fd, unsigned char *digest, int len)
{
    md5_msg_t msg;

    if (len < 16)
        { errno = EINVAL;
        return -1;
    }

    msg.msgtype = MD5_RECV_DIGEST;
    msg.msglen = len;
    return MsgSend(fd, &msg, sizeof(msg), digest, len);
}
```

Of course, as a resource manager developer, you would package the `md5_send()`, `md5_recv()` functions into a library for others to use your service.

```
int main(int argc, char **argv)
{
    if ((fd = name_open("md5name", 0)) == -1)
        { perror("name_open");
        return -1;
    }

    if ((cfd = open(argv[1], O_RDONLY)) == -1)
        { perror("open");
        return -1;
    }

    total = 0;

    for (;;) {
        n = read(cfd, buf, 16 * 1024);
        if (n <= 0)
              break;

        if (md5_send(fd, buf, n) <= 0)
              { perror("md5_send");
              return -1;
```

```
            }

            total += n;
        }

        md5_recv(fd, digest, 16);
        printf("%10d\t\t\t", total);
        for (n = 0; n < 16; n++) {
            printf(" %02X", digest[n]);
        }
        printf("\n");
        return 0;
    }
```

It seems like everything is done, but is there a problem?
Firstly, this server can only handle one request at a time. If two programs request this service simultaneously, there is a logical error. Moreover, this solution implies that every client who wants to use this MD5 service must link to a dedicated library. Is there no way to make it more universal?
The answer is yes. Since the MD5 service is provided through a name, and POSIX has a standard definition for operations that can be performed on files, for our service, it immediately comes to mind that the client can directly write() data to, for example, /dev/md5, and then read() the result.

So, how should the resource manager behave if the client writes? What information will I receive? QNX has already prepared for you, and this is the resource manager framework.

Resource Manager Framework

Since a resource manager is entered through a path name, and POSIX has defined operations that can be performed on a file, QNX has predefined the message types and data formats required for these operations.
The resource manager framework provided by QNX can be roughly divided into four parts:

- The `iofunc` (iomsg) layer, which provides all POSIX file IO operations (sys/iofuncs.h, sys/iomsg.h)
- The resmgr layer, which registers the path name, receives data, and distributes it to iofunc to execute specific operations. iofunc and resmgr are the foundations for writing a resource manager.
- dispatch layer, in some complex resource managers, "incoming message passing" is not the only thing that needs to be handled. There may also be a need to handle "pulses" or sometimes a "signal". The dispatch layer proactively identifies different types of input information and then forwards them to different handling functions for processing.
- thread pool layer, this layer provides thread pool management, which can be configured to implement multiple threads for resource management.

Let's take a deeper look at each layer.

iofunc (iomsg) layer:

QNX has summarized a total of 34 file operations; essentially, all POSIX file handling can be conducted through these 34 operations. The iofunc layer of the resource manager is essentially about preparing callback functions that provide services by responding to these operation requests.

These 34 callback functions are divided into 8 "connect" callback functions and 26 "io" callback functions, based on their nature. These functions are all defined in io_func.h, and they are as follows:

Connect Functions	IO Functions
open	read
unlink	write
rename	close_ocb
mknod	stat
readlink	notify
link	devctl
unblock	unblock
mount	pathconf
	lseek
	chmod
	chown
	utime
	openfd
	fdinfo
	lock
	space
	shutdown
	mmap
	msg
	umount
	dup
	close_dup
	lock_ocb
	unlock_ocb
	sync
	power

Each callback function in the iofunc layer has a corresponding data structure for message passing between the client and the server, which is defined in sys/iomsg.h.
For example, for io_read, there is a corresponding structure:

```
typedef union {
      struct _io_read i;
      /* unsigned char data[nbytes]; */
} io_read_t;
```

It can be seen that for `io_read`, there is a `struct _io_read I;` that is sent from the client to the server. However, from the server back to the client, there is no special message structure; the data that is read is simply returned directly.
As for `io_stat`, it would be something like:

```
typedef union {
      struct _io_stat    i;
      struct  stat         o;
} io_stat_t;
```

You can see that `io_stat` sends a `struct _io_stat` from the client to the server; whereas what the server returns to the client is a `struct stat o;`. This `struct stat` is the return value obtained by the POSIX `stat()` function.
If you observe the data structures such as `struct _io_read` and `struct _io_stat` closely, you will notice that they all begin with:

```
{
      _Uint16t  type;
      _Uint16t  combine_len;
      …
}
```

while "combined_len" notifies the server of the total length of this message (including any variable-length parameters it might carry).

The definition of "type" is already established in sys/iomsg.h, for example, "_IO_READ", "_IO_WRITE", "_IO_STAT", and so on.

We have explained how a read() function sends data in another article before. Let's take another look

```
ssize_t read(int fd, void *buff, size_t nbytes)
      { io_read_t  msg;

      msg.i.type = _IO_READ;
      msg.i.combine_len = sizeof msg.i;
      msg.i.nbytes = nbytes;
      msg.i.xtype = _IO_XTYPE_NONE;
      msg.i.zero = 0;
      return MsgSend(fd, &msg.i, sizeof msg.i, buff, nbytes);
}
```

So, in comparison with the data structures in iomsg.h, it is not difficult to imagine how those POSIX standard file functions in libc are constructed, such as write(), stat(), lseek(), and so on.

All programs on QNX call these functions in libc, and within these functions, they are all converted into individual messages and sent to the relevant resource servers through file descriptors (fd).

Ordinary developers on QNX simply call these read(), write(), stat() functions, then get the results without worrying about the internal implementation. This is also why many basic Unix programs can be easily recompiled on QNX.

On the resource server side, based on the type in iomsg, it can be determined what kind of request it is, and then the corresponding handler in iofunc is called to process it.

Maybe you've already thought of the next step of this issue. Writing a resource manager requires preparing dozens of callback functions, which is overly complex. Moreover, many times, a resource manager only provides a service through a pathname and does not need to offer all the POSIX standard processing. For example, /dev/random is a service that provides random numbers, and there is no need to lseek() it, right? So, from the resource manager's perspective, is it possible not to provide the IO_SEEK callback function code?

The answer is, of course, yes, and this introduces the resmgr layer.

resmgr layer

If the iofunc layer provides various callback functions, then the resmgr layer is the one that "loops to receive information and calls iofunc". Here are some of the key functions involved.

```
resmgr_attach(….)
ctp = resmgr_context_alloc()
for (;;) {
      ctp = resmgr_block(ctp);
      resmgr_handler(ctp)
}
```

Combining the resmgr layer and the iofunc layer allows us to build a resource manager. Let's look at a simple example.

This time, we intend to write a resource manager for "/dev/now". When someone reads it, it will return the current time as a string. Let's take a look at the code:

```
/* Create a dispatcher, think this as a Channel to receive data */
dispatch = dispatch_create();


/* Some attribute of resource manager itself. */
memset( &res_attr, 0, sizeof( res_attr ) );
res_attr.nparts_max = 10;
res_attr.msg_max_size = 0;

/* think io_attr as attribute of file, like read/write permission */
iofunc_attr_init(&io_attr, 0666 | S_IFCHR, 0, 0);
```

```
/* Initialize iofunc call back layer*/
iofunc_func_init( _RESMGR_CONNECT_NFUNCS, &connect_funcs,
             _RESMGR_IO_NFUNCS, &io_funcs );

/* Build the resource manager layer, and register the path */
rmgid = resmgr_attach(dispatch, &res_attr, "/dev/now", _FTYPE_ANY, 0,
                  &connect_funcs, &io_funcs,&io_attr);

/* Prepare a resource manager context */

ctp = resmgr_context_alloc(dispatch);
while (1)
{
      /* resmgr_block(), this is like a MsgReceive() */
      ctp = resmgr_block(ctp);

      /* Based on what has been received, call the callback layer*/
      resmgr_handler(ctp);
}
```

Once encapsulate the provided code in a `main()` function, you will find there is a "/dev/now" folder in the system.

```
$ ls -la /dev/now
crw-rw-rw-   1 root       root         0,  1 Jun 01 01:46 /dev/now
```

This service, because we haven't implemented any `iofunc` callback functions yet, naturally won't provide any services.

But wait a moment? How does *ls -la* succeed then? Essentially, *ls -la /dev/now* does the following:

```
fd = open("/dev/now", O_RDONLY);
stat(fd, &mystat);
close (fd)
printf()
```

In other words, resmgr has already implemented at least the io_open(), io_stat(), and io_close() callback functions for us, right? And the permissions string "crw- rw-rw-" is actually due to the parameters we provided when initializing io_attr.

Let's add a callback function to implement the functionality of /dev/now:

```
static int now_read(resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T
*ocb)
{
      iofunc_ocb_t *o = (iofunc_ocb_t *)ocb;
      char nowstr[128];
      time_t t;
      int n;
```

```
        /* Take current time, write it into a string */
        t = time(NULL);
        n = strftime(nowstr, 128, "%Y-%m-%d %H:%M:%S\n", localtime(&t));

        /* Return the string to client */
        MsgReply(ctp->rcvid, n, nowstr, n);

        /* Tell resmgr layer, we already did the MsgReply, don't do reply again */
        return _RESMGR_NOREPLY;
}
```

Then, add this callback function during resmgr initialization:

```
iofunc_attr_init(&io_attr, 0666 | S_IFCHR, 0, 0);
iofunc_func_init( _RESMGR_CONNECT_NFUNCS, &connect_funcs,
                            _RESMGR_IO_NFUNCS, &io_funcs );

io_funcs.read = now_read;
```

Now, execute the devnow program to start the /dev/now service:

```
$ cat /dev/now
2008-09-12 10:23:06
2008-09-12 10:23:06
2008-09-12 10:23:06
2008-09-12 10:23:06
2008-09-12 10:23:06
2008-09-12 10:23:06
…
```

Although we correctly get the time displayed, why does it keep displaying continuously? This is because the cat command works by looping read() until it reaches the end of the file. So how do we tell the user program that "we have reached the end of the file"? Essentially, read() returning a length of 0 indicates this. In other words, in our io_read() callback function, we need to distinguish between the first read() (which returns the time string) and subsequent read() (which returns length 0). We can do this by keeping track of the file read offset. During the first read, the offset is 0, but for subsequent reads, the offset will not be 0.

Let's look at the modified code below.

```
static int now_read(resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T
*ocb)
{
        iofunc_ocb_t *o = (iofunc_ocb_t *)ocb;
        char nowstr[128];
        time_t t;
        int n;

        /* Determine if it's the first read() after opening the file; if not, reply
        with 0. This way, the client's read() will return 0 to indicate reaching
        the end of the file. */

        if (o->offset != 0) {
                return 0;
        }
```

```
        /* Get the current time and write it to a string */
        t = time(NULL);
        n = strftime(nowstr, 128, "%Y-%m-%d %H:%M:%S\n", localtime(&t));

        o->offset += n;

        /* Return the string to the client */
        MsgReply(ctp->rcvid, n, nowstr, n);

        /* Tell the resmgr layer that we have already replied, so it should not
        reply to the client again */
        return _RESMGR_NOREPLY;
}
```

Now, if we cat again:

```
$ cat /dev/now
2008-09-12 10:25:28
$
```

The full source code of devnow can be found in Github

iofunc_default_ callback function

If a resource manager do not need to present standard POSIX file attribution, then like above sample, things are simple. But if resource manager need to support full POXI file, things cecome more complex.

POSIX file handling involves several important internal connections. For instance, in the io_open() callback, it's necessary not only to check if the client has the correct read and write permissions but also to remember the file's open mode and reflect this in future callbacks (clearly, a client that opens a file with O_RDONLY cannot perform io_write()). The file manager also needs to continuously track read and write positions to ensure that future lseek() / tell() calls return correct values. Additionally, every read() / write() operation requires updating attributes in the stat structure such as st_mtime and st_atime, among others.

To assist developers in handling POSIX files correctly, QNX provides iofunc_default_* functions in the iofunc layer.

```
iofunc_open_default()
iofunc_chmod_default()
iofunc_devctl_default()
…
```

Still as resource manager sample, we add a few lines to register /posix_file file:

```
dispatch = dispatch_create();

memset( &res_attr, 0, sizeof( res_attr ) );
res_attr.nparts_max = 10;
res_attr.msg_max_size = 0;
```

```
iofunc_attr_init(&io_attr, 0666 | S_IFCHR, 0, 0);
iofunc_func_init( _RESMGR_CONNECT_NFUNCS, &connect_funcs,
                  _RESMGR_IO_NFUNCS, &io_funcs );
connect_funcs.open = iofunc_open_default;
io_funcs.read          = iofunc_read_default;
io_funcs.write         = iofunc_write_default;
io_funcs.chmod         = iofunc_chmod_default;
io_funcs.chown         = iofunc_chown_default;
io_funcs.stat          = iofunc_stat_default;
io_funcs.close_ocb     = iofunc_close_ocb_default;

rmgid = resmgr_attach(dispatch, &res_attr, "/posix_file", _FTYPE_ANY,
                      0, &connect_funcs, &io_funcs, &io_attr);

ctp = resmgr_context_alloc(dispatch);
while (1)
{
      ctp = resmgr_block(ctp);
      resmgr_handler(ctp);
}
```

Detail code is in Github, in this resource manager we can do some operation:

```
# ls -lc /posix_file
crw-rw-rw-  1 root      root          0,   1 Jun 05 08:34 /posix_file
```

You can see file's owner is root, and mode is 666. File's update time is 08:34

```
# chown xtang /posix_file
# chmod 777 /posix_file
# echo "hello" >/posix_file
# ls -lc /posix_file
crwxrwxrwx  1 xtang     root          0,   1 Jun 05 08:35 /posix_file
```

You can see file's mode, owner, update time are all correctly changed.

In this example code, we directly attached functions like iofunc_chmod_default and iofunc_chown_default, which can handle standard POSIX file commands, but in reality, they do not provide any functionality. In practical scenarios, it's common to attach our own functions to handle resource manager operations and then invoke iofunc_*_default functions to handle related POSIX operations. This means...:

```
iofuncs.write = my_write;
int my_write(resmgr_context_t *ctp, io_wriet_t msg, iofunc_ocb_t *ocb)
{
      /* do special operation */

      return iofunc_write_default(ctp, msg, ocb);
}
```

Sure, let's outline how we can rewrite our MD5 server using iofunc and resmgr. We'll focus on implementing the io_write() and io_read() callback functions. When the client writes data using write(), we'll continuously feed the incoming data into MD5_Update(). Then, when the client calls read(), we'll compute the current digest using MD5_Final() and return it to the client.

There is a fundamental issue here. In MD5 computation, there is a data structure called MD5_CTX, which is initialized using MD5_Init() and then continuously updated with data using MD5_Update(). Finally, when read() is called, you obtain the digest returned by MD5_Final().

The question arises: these io_read() and io_write() functions are callback functions. How can we ensure that the MD5_CTX structure persists consistently across multiple incoming write() and read() operations?

The answer lies in extending the OCB (Open/Close/Bind) structure.

Extende OCB

OCB stands for Open Context Block. When a file is opened, an OCB is prepared. This OCB is associated with the file descriptor (fd), and as long as the file is not closed, any iofunc callback on the same fd will receive this OCB. If multiple fds are opened simultaneously, each fd will have its own unique OCB. Clearly, if we embed our md5_context_t within the OCB, each fd will have its own MD5_context_t. How can we achieve this?

First, define an extended OCB structure as follows

```
/* extend iofunc_ocb_t */
typedef struct {
      iofunc_ocb_t ocb;
      int          total_len;
      MD5_CTX      md5_ctx;
} md5mgr_ocb_t;
```

Please note, the first element in the structure must always be iofunc_ocb_t. This ensures that our extended structure can be directly used as iofunc_ocb_t operations. If needed, we can still invoke iofunc_*_default functions to handle POSIX-related defaults. The total_len and md5_ctx following this structure are the extensions.

Next, how should we allocate memory for this extended structure? There are two ways to do this. One way is to replace the ocb_calloc() callback function in the iofunc layer.

```
iofunc_funcs_t ocb_funcs = {
      _IOFUNC_NFUNCS,
      md5mgr_ocb_calloc,
      md5mgr_ocb_free
 };
iofunc_mount_t mountpoint = { 0, 0, 0, 0, &ocb_funcs };
iofunc_attr_init(&io_attr, 0666 | S_IFCHR, 0, 0);
io_attr.mount = &mountpoint;
```

Another more direct approach is to prepare our own io_open() callback function. When the client calls open(), it will enter our callback. Inside this callback, we can malloc() our own structure, and

then bind this structure to the fd using resmgr_bind(), treating it as the OCB. This way, this OCB structure will be passed into all subsequent IO callbacks.

The complete code for this new MD5 service can be found under md5mgr on GitHub

dispatch layer

Although in most cases, using `iofunc` layer + `resmgr` layer is sufficient to construct a complete resource manager, sometimes a resource manager needs to handle other types of messages concurrently. This is where the dispatch layer comes in. The dispatch layer can handle various forms of messages. In addition to attaching `resmgr` and `iofunc` using `resmgr_attach()`, it can also do the following (sys/dispatch.h):

message_attach()

```
int message_attach(dispatch_t *dpp, message_attr_t *attr, int low, int high,
                   int (*func)(message_context_t *ctp, int code, unsigned flags, void *handle), void
                   *handle);
```

Sometimes a resource manager, besides using standard iomsg, also wants to define its own message types. This is where message_attach() comes into play. This function means that if a message is received with a type between low and high, then func is called to handle it.

Of course, it's important to ensure that low and high do not overlap with iomsg, otherwise there would be ambiguity. All iomsg types are defined between _IO_BASE and _IO_MAX in iomsg.h, so as long as low > _IO_MAX, this condition is satisfied.

pulse_attach()

```
int pulse_attach(dispatch_t *dpp, int flags, int code,
                 int (*func)(message_context_t *ctp, int code, unsigned flags, void *handle),
                 void *handle);
```

Many resource managers managing hardware resources (device drivers), in addition to providing `iomsg` messages to handle client I/O requests, often need to receive 'pulses' to process interrupts (InterruptAttachEvent). In such cases, `pulse_attach()` can be used to bind a specified pulse code to a callback function `func`.

select_attach()

```
int select_attach(void *dpp, select_attr_t *attr, int fd, unsigned flags,
                  int (*func)(select_context_t *ctp, int fd, unsigned flags, void *handle),
                  void *handle);
```

Many resource managers, when handling client requests, also need to send requests to other resource managers. Often, these requests may not return results immediately. Typically, we could handle this using select(), but firstly, we cannot use a blocking select() because we are a service function of a resource manager—if blocked, we can't return, meaning we can't process client requests. Secondly, we can't use select() in a polling manner because once we return, we enter a blocking state waiting for client messages; without new messages, we won't exit the blocking state, losing the opportunity to poll select() again.

Of course, you could set your own timer, sending yourself a pulse periodically to wake up and then poll select() again. It's feasible but unnecessarily increases system overhead.

select_attach() is designed for this purpose. For a specific file descriptor (fd), the unsigned flags here determine the select() events you're interested in (Read? Write? Except?). This means that if the selected flags events occur for the fd, it calls the func callback function.

When using dispatch, after attaching with special *_attach() calls, you simply replace several functions in the resource manager layer with functions from the dispatch layer, like this...

```
ctp = dispatch_context_alloc(dispatch);
while (1)
{
        ctp = dispatch_block(ctp);
        dispatch_handler(ctp);
}
```

dispatch_block() functions similarly to blocking and waiting, while dispatch_handle() calls different callback functions based on different attachments for processing.

Another commonly used dispatch function is dispatch_create_channel(). Sometimes, you may prefer to create a channel yourself using ChannelCreate() instead of letting dispatch_create() automatically create it for you. This function allows you to create channels yourself because sometimes you want to set specific flags (flags in ChannelCreate()) on the channel.

thread pool layer

The resource managers described above, which use while (1) loops to handle messages, are clearly 'single-threaded' resource managers, utilizing only one thread to process client requests.

For resource managers that need to handle client requests frequently, the natural consideration is to use a 'multi-threaded' resource manager. Essentially, this involves using several threads to execute the while (1) loop described earlier.

A Thread Pool is used to achieve this. It's relatively straightforward to use: first, configure pool_attr, then create and start the thread pool.

```
memset(&pool_attr, 0x00, sizeof pool_attr);
if(!(pool_attr.handle = dpp = dispatch_create())) {
      perror("dispatch_create");
      return EXIT_FAILURE;
}
pool_attr.context_alloc = dispatch_context_alloc;
pool_attr.block_func    = dispatch_block;
pool_attr.handler_func  = dispatch_handler;
pool_attr.context_free  = dispatch_context_free;
pool_attr.lo_water      = 2;
pool_attr.hi_water      = 5;
pool_attr.increment     = 2;
pool_attr.maximum       = 10;

tpp = thread_pool_create( &pool_attr, POOL_FLAG_EXIT_SELF)
thread_pool_start( tpp );
```

The earlier callbacks for pool_attr are relatively simple. Here's a brief explanation of lo_water, hi_water, increment, and maximum:

- maximum indicates the maximum number of threads that can be created in the pool.
- hi_water specifies the maximum number of these threads that can wait for tasks.
- lo_water indicates the minimum number of threads that should be waiting for tasks.
- increment determines the number of threads incremented at a time

Using the example above, once this thread pool is started, it initially creates 5 threads all waiting on dispatch_block() to receive tasks.

When a request arrives, Thread 1 handles it. If during its service, Thread 1 needs to request data from another thread and momentarily cannot return, there will be 4 threads remaining waiting for tasks.

If two more requests arrive, we have 3 threads servicing requests with 2 threads waiting for tasks.

When the 4th request arrives and another thread begins servicing it, leaving only 1 thread waiting for tasks, which is less than lo_water, the thread pool automatically creates 2 additional threads (increment) and places them in the waiting task queue. So now we have 4 threads servicing requests and 3 threads waiting.

If service threads cannot be reclaimed and new requests arrive, causing the number of waiting threads to fall below lo_water, the thread pool will continue to increase threads to ensure there are enough waiting threads for service. However, the total number of service threads and waiting threads combined will not exceed maximum, which is 10 in this case.

If, in a state where 4 threads are servicing requests and 3 are waiting, 2 service threads finish their tasks and return to the pool, the thread pool will put these 2 threads back into the waiting queue. This means there will be 2 service threads and 5 waiting threads.

Now, if another service thread finishes and returns to the pool, if this thread is put back into the waiting queue, there will be 6 threads waiting for service, which exceeds hi_water. Therefore, the thread pool will terminate this thread. Now, we have 1 thread servicing requests and 5 waiting, reducing the total threads to 6.

In summary, using a thread pool allows dynamic and flexible configuration of multi-threaded resource managers. When many requests come simultaneously, threads are immediately deployed for service. When there are fewer idle threads, the thread pool preempts by starting additional threads in advance for potential service needs. After service completion, the thread pool also shuts down surplus threads accordingly.

In conclusion

You can see the importance of resource managers on QNX; almost all services are implemented through resource managers. Using the concept of resource managers, systems can be modularized effectively. Therefore, a correct understanding of resource manager concepts and proficient use of the resource managers provided by QNX are crucial skills for development on QNX.

Of course, resource managers also have their weaknesses. Typically, a manager needs to collaborate with others to complete system functions. In such cases, it's crucial to ensure that single-threaded managers do not get blocked and unable to provide services. While multi-threaded managers are not as concerned about blocking, careful attention to thread synchronization is essential.

Additionally, a critical factor in system design lies in properly designing the segmentation of resource management. If segmented too finely, a task might need to pass through multiple resource managers, severely impacting performance. If segmented too coarsely, modular division is lost, potentially turning into a complex system and increasing the difficulty of debugging and error management.