QNX® Neutrino® Architecture

QNX SOFTWARE SYSTEMS

NOTES:

You will learn:

- the architecture of the QNX Neutrino RTOS
  - how it's different from others
  - what this means
- operating system services and what delivers them
- process and thread models
- how scheduling works

NOTES:

Topics:

NOTES:

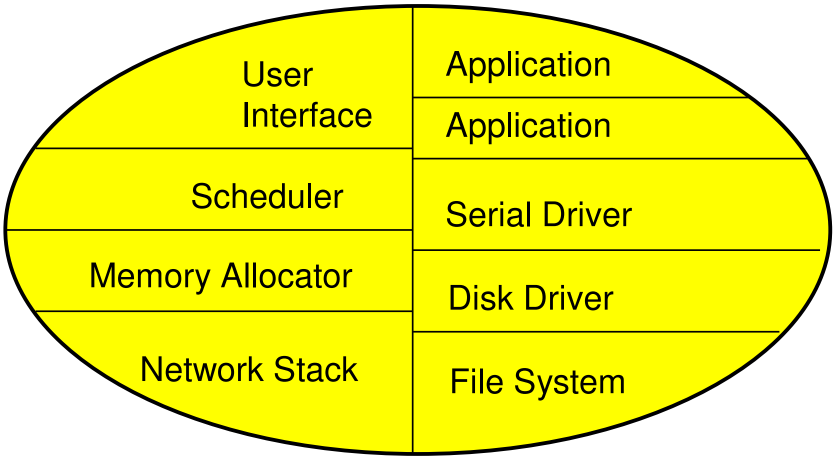
Portability

QNX Neutrino delivers a standards based system in a small form factor:

- POSIX 1003.1-2001
  - Unix, threads, timers, signals, etc
- ANSI C/C++
  - GNU Compiler Chain

NOTES:

In a traditional Real-Time Executive:

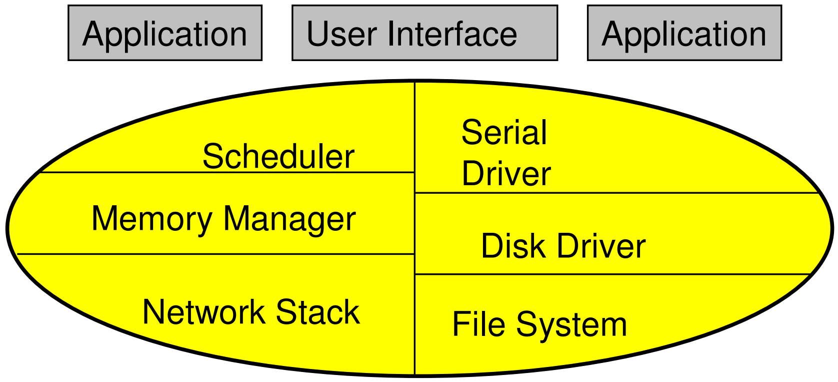


– All modules share the same address space and are, effectively, one big program.

NOTES:

Architecture - Monolithic

In a traditional Monolithic kernel OS:



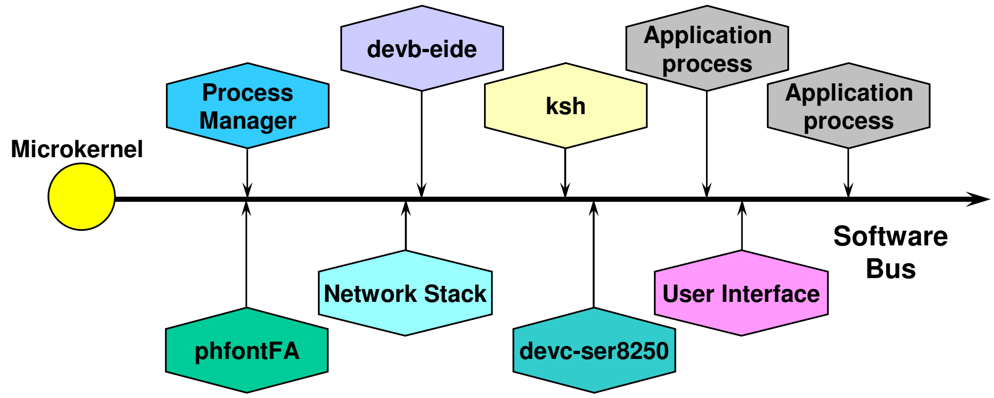
Monolithic Kernel

- The kernel contains the OS kernel functionality and all drivers, so driver development is complex and debugging can be painful.
- Applications are processes in protected memory space, so the kernel is protected from applications and applications are protected from each other.

NOTES:

The best known is the many different flavours of Unix.

In the QNX Neutrino OS:



- The OS consists of the microkernel (or just “kernel”) and a set of cooperating processes.
- The processes are separate from the kernel so if something goes wrong in a process it would not affect the kernel.

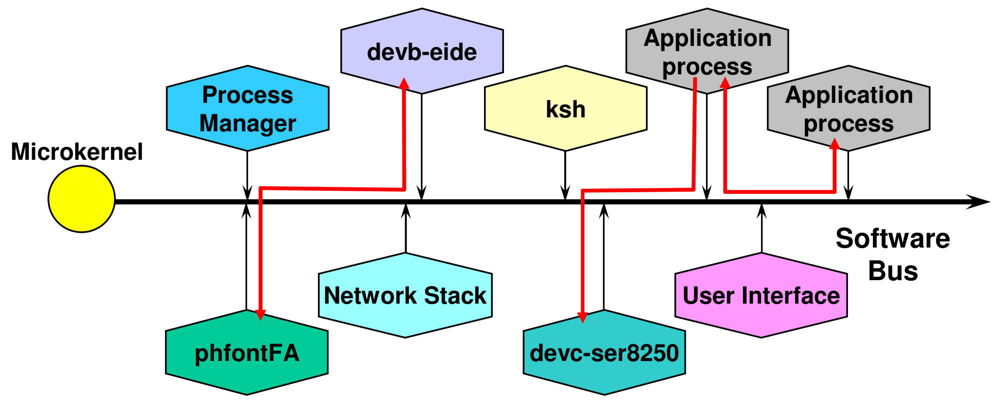
NOTES:

Drivers are just processes, so that the kernel is even protected from driver problems and drivers are protected from each other. Drivers can be started, stopped, and debugged like any other process.

There is one exception though. The driver may contain an interrupt handler and if something goes wrong in the interrupt handler then it could bring down the kernel. However, in QNX most, if not all, of the interrupt handling is usually done outside of the interrupt handler.

QNX SOFTWARE SYSTEMS

Processes communicate with each other:



- the OS processes and your processes cooperate using interprocess communication. Together, the OS and your processes make up one seamless system.
- there are a large variety of types of interprocess communication

NOTES:

In the above diagram:

- the font manager process, phfontFA, needed the data for a particular font so it found and communicated with the disk driver, devb-eide, in order to get that data from the disk,
- an application process is writing to the serial port. It does this by communicating with the serial port driver, devc-ser8250,
- one application process is communicating with another application process.

Examples of processes are:

- Disk Drivers
  devb-eide, devb-aha2
- Network Stack
  io-pkt
- Character Drivers
  devc-ser8250, devc-serppc800, devc-con
- GUI components
  Photon, phfontFA, io-graphics
- Bus managers
  pci-raven, devp-pccard
- System daemons
  cron, inetd, mqueue, qconn

NOTES:

If you want to see what processes are running on your system:

In the QNX IDE we use:

The Target Navigator view, usually found in the System Information Perspective

From a command line do:

`pidin`

So what does this mean?

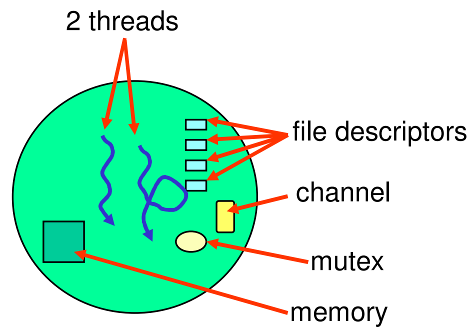
Trade-offs:

- benefits:
  - resilience and reliability
  - ease of configuration and reconfiguration
  - ease of debugging
  - ease of development
  - scalability
- costs:
  - system overhead
    - more context switches
    - more copies of data

NOTES:

What is a process?

- a program loaded into memory
- identified by a process id, commonly abbreviated as **`pid`**
- owns resources:
  - memory, including code and data
  - open files
  - identity - user id, group id
  - timers
  - and more



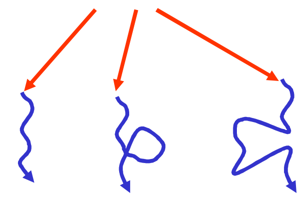
Resources owned by one process are protected from other processes

NOTES:

What is a thread?

- a thread is a single flow of execution or control
- a thread has some attributes:
  - priority
  - scheduling algorithm
  - register set
  - CPU mask for SMP
  - signal mask
  - and others
- all its attributes have to do with running code

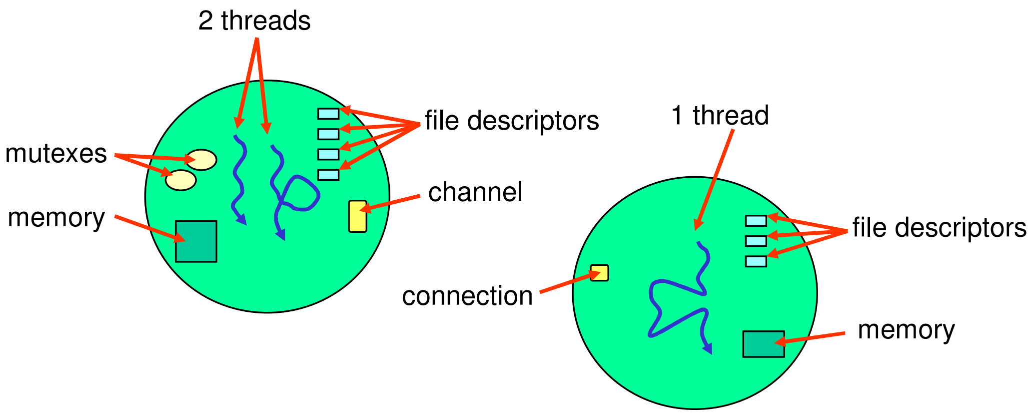
3 threads



NOTES:

Threads run in a process:

- a process must have at least one thread
- threads in a process share all the process resources



Threads run code, processes own resources

NOTES:

Processes and threads:

- processes are your "building blocks" components of a system
  - visible to each other
  - communicate with each other
- threads are the implementation detail
  - hidden inside processes

NOTES:

Topics:

- Overview
- → The Microkernel
- The Process Manager
- Scheduling
- Adaptive Partitioning
- SMP
- Resource Managers
- System Library
- Shared Objects
- OS Services
- Conclusion

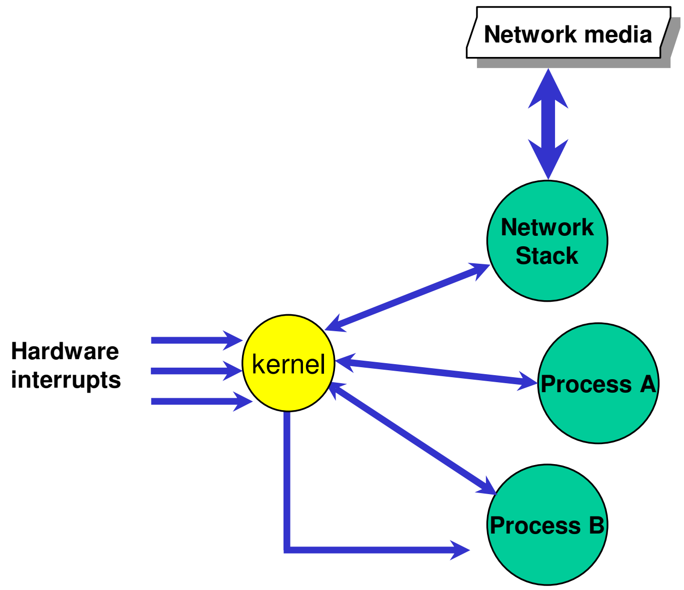
NOTES:

The kernel is special:

- it is the glue that holds the system together
- programs deal with the kernel by using special library routines, called "kernel calls", that execute code in the kernel
- most of the other sub-systems, including user applications, communicate with each other using the message passing provided by the kernel through kernel calls

NOTES:

The kernel is the core of your system:



NOTES:

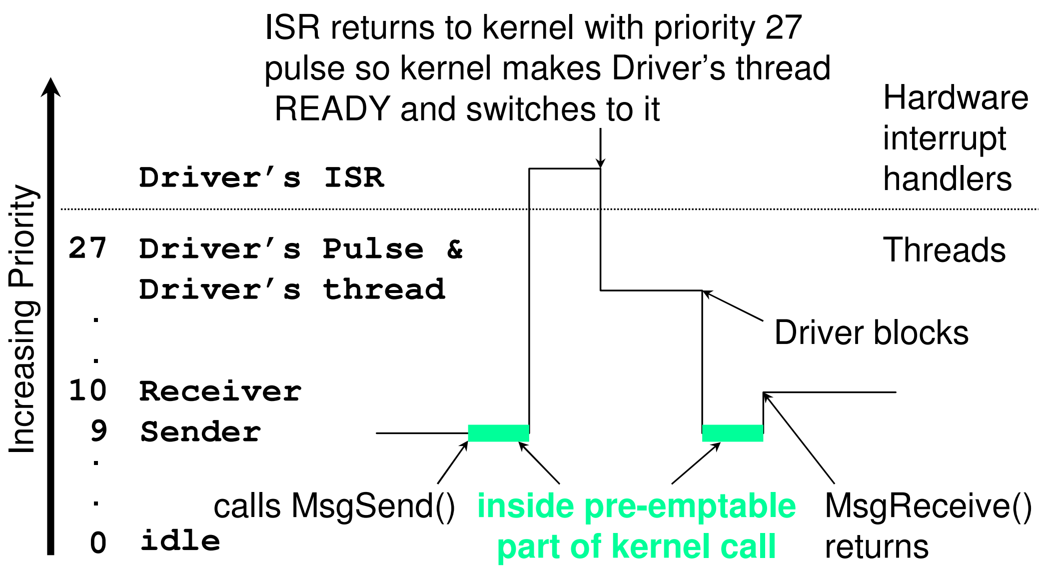
Kernel calls:

- often you'll make kernel calls
- this means you'll be executing code in the kernel for the duration of the call
- what if a time critical event occurs?

NOTES:

Kernel - Pre-emption

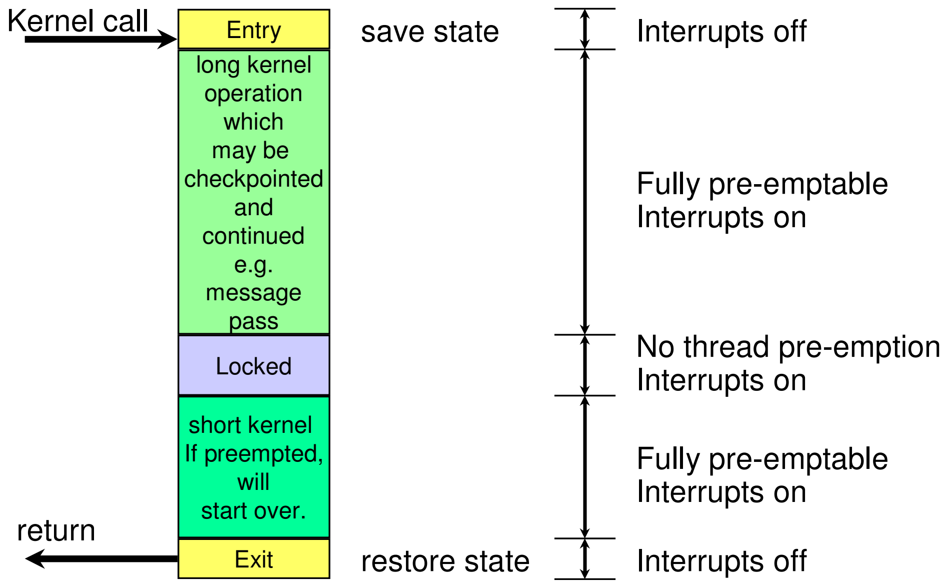
Kernel calls are pre-emptable:



NOTES:

Kernel - Pre-emption

Kernel operations:



NOTES:

This is not a time-ordered diagram, the internal states can be intermixed.

Long kernel operations, such as copying the data during a message pass, will have their current state saved when pre-empted, and the operation will continue from where they got pre-empted.

For some very short operations, such as thread state changes or mutex locking, pre-emption will be disabled since the operation has to complete atomically. This is the Locked section shown above.

Short kernel operations, such as validating addresses for a *MsgSend()*, will be restarted if they are pre-empted. This is generally used for operations where the cost of checkpointing is comparable to the cost of the operation.

Most kernel calls will involve a combination of the above operations and states.

The kernel call and return (kernel entry/exit) will be implemented with whatever works best on a particular architecture, whether it be software interrupt, sysenter/sysexit routines, or something else.

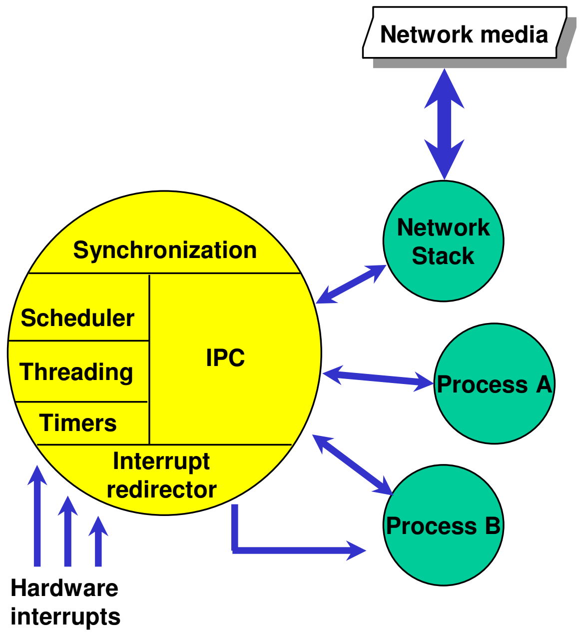
So what does this mean?

More trade-offs:

- benefit: reduce latency
  - respond to new events faster
  - shorter interrupt latency, scheduling latency
- cost: throughput
  - takes more time to restart an interrupted kernel call
  - take more time to save current state & restart a pre-empted message pass

NOTES:

The kernel provides:



NOTES:

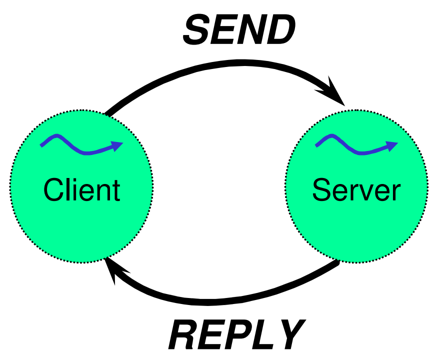
The forms of IPC provided by the kernel:

- Messages
  - exchanging information between proceses
- Pulses
  - delivering notification to a process
- Signals
  - interrupting a process and making it do something different (usually termination)

NOTES:

QNX SOFTWARE SYSTEMS

Native QNX Neutrino Messages:

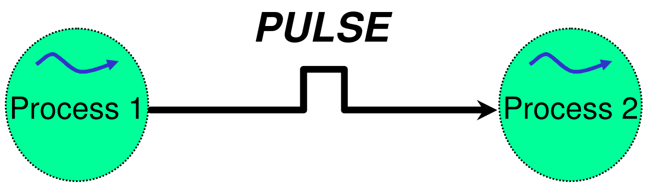


NOTES:

The client/server model shown here is the basis for QNX Neutrino's great flexibility -- almost all installable components, including the POSIX message queue support, file system, etc, use this model as the basis for communication.

Native QNX Neutrino Pulses:

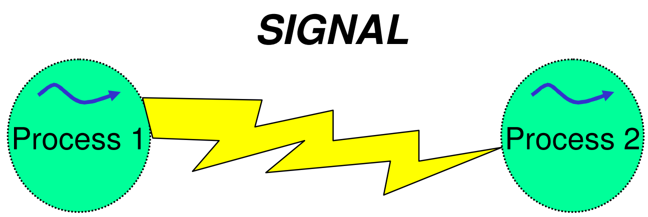
- used for event notification: “something happened”



NOTES:

POSIX Signals:

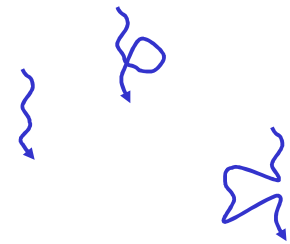
– interrupt another process



NOTES:

Thread Functions:

- create / terminate threads
- wait for thread completion
- change thread attributes



NOTES:

Thread synchronization methods:

mutex	mutually exclude threads
condvar	wait for a change
semaphore	wait on a counter
join	synchronize to termination of a thread

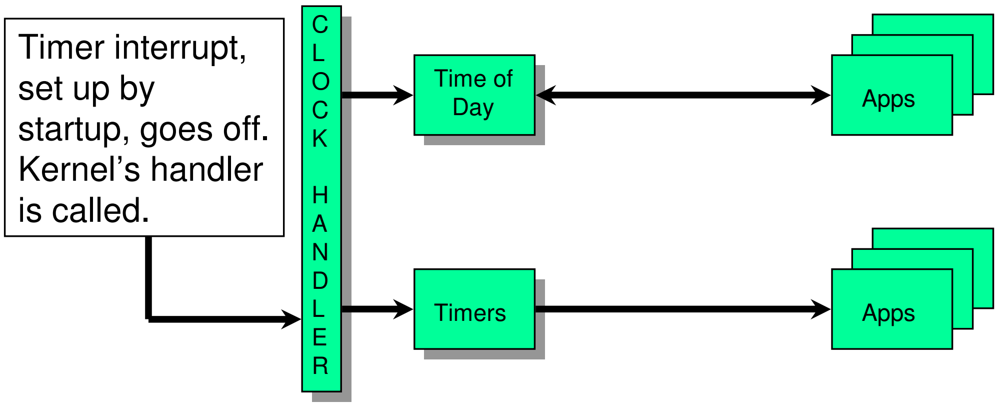
NOTES:

Note that with mutexes and condition variables, we can also synchronize across process boundaries by using shared memory.

Semaphores are available in two flavors -- unnamed (which are provided by the kernel itself), and named (which require an additional resource manager).

Further synchronization methods, including Barriers, Rwlocks, and sleepons are provided by building on top of mutexes and/or condvars.

QNX® Neutrino®'s Concept of Time:

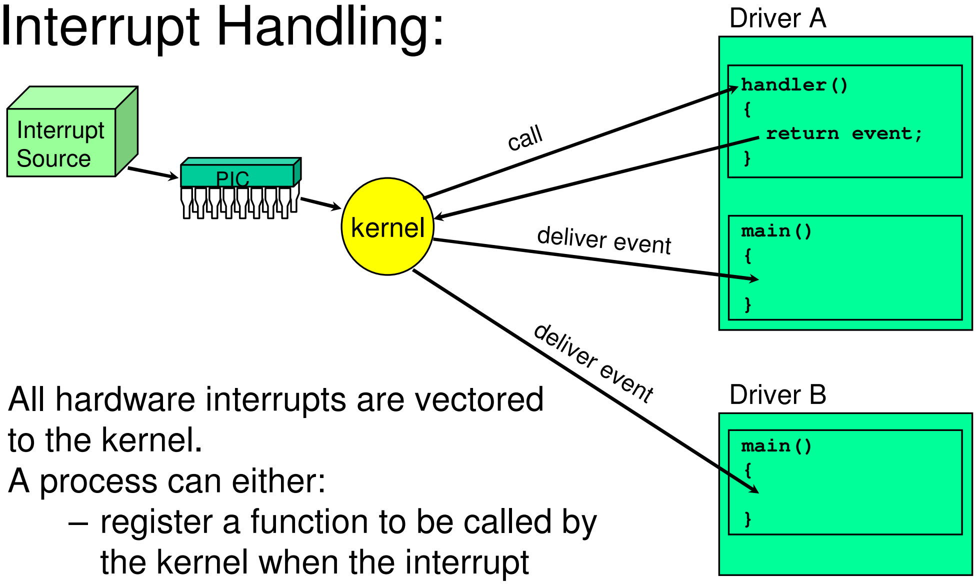


NOTES:

The kernel:

- keeps track of the current time of day which you can get and set
- does time accounting for thread CPU times and scheduling decisions (RR/Sporadic)
- provides interfaces for:
  - periodic or one-shot timers
  - timeouts for blocking calls

Interrupt Handling:



All hardware interrupts are vectored to the kernel.

A process can either:

- register a function to be called by the kernel when the interrupt happens
- request notification that the interrupt has happened

NOTES:

The kernel:

- can be thought of as a library
  - no processing loop, no `while(1)`
- only runs if invoked by:
  - kernel call
  - interrupt
  - processor fault/exception e.g. illegal instruction, invalid address

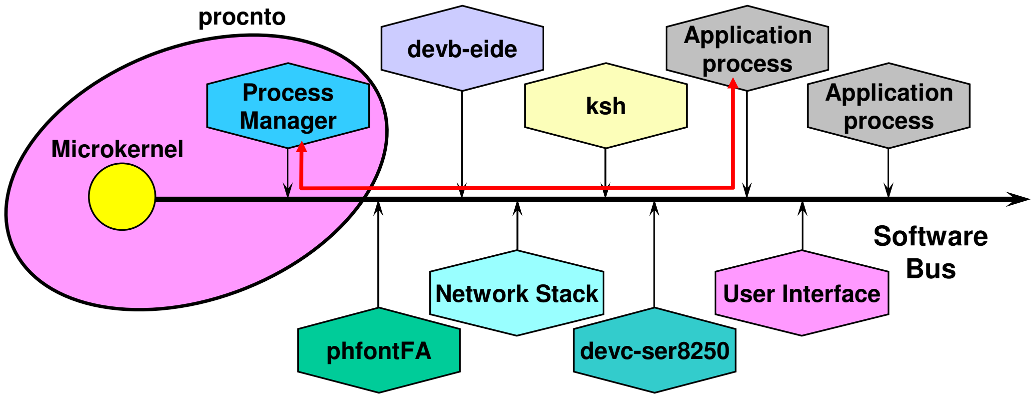
NOTES:

Topics:

- Overview
- The Microkernel
- → The Process Manager
- Scheduling
- Adaptive Partitioning
- SMP
- Resource Managers
- System Library
- Shared Objects
- OS Services
- Conclusion

NOTES:

Communication with the Process Manager:



- **procnto** is QNX
  - –proc for the process manager
  - –nto for the Neutrino microkernel
  - –they share address space, but behave differently
- process manager is reached using messages

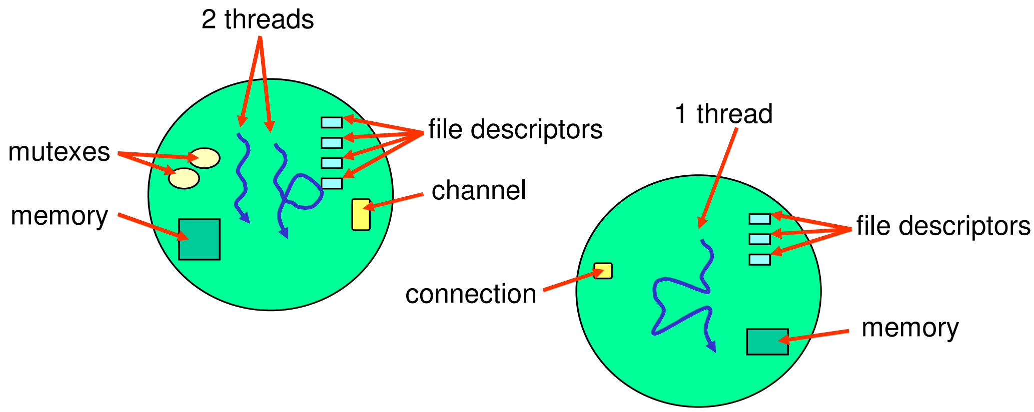
NOTES:

The Process Manager provides:

- packaging of groups of threads together into processes
- memory protection, address space management including shared memory for IPC
- pathname management
- process creation and termination
  - spawn / exec / fork
  - loads ELF executables
- an idle thread that uses CPU no-one else wants

NOTES:

Each process is a collection of resources and one or more threads:



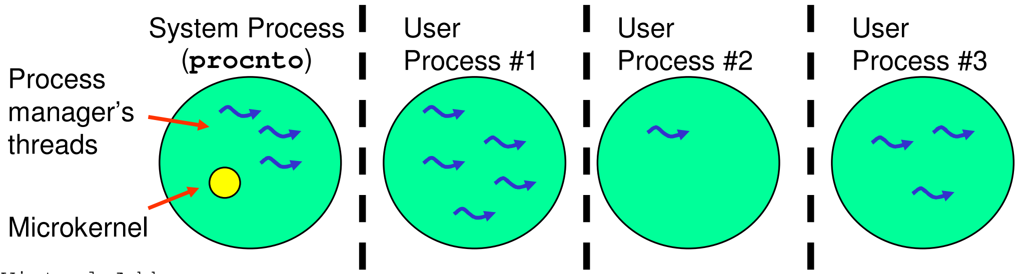
– let's look at memory in more detail…

NOTES:

QNX SOFTWARE SYSTEMS

We use a virtual address model:

– each process runs in its own protected virtual address space
– pointers that you deal with contain virtual addresses, not physical
– physically they all share the same address space



Virtual Addresses

	System Process	User Process #1	User Process #2	User Process #3
x86:	3.5G ↔ 4G	0 ↔ 3.5G	0 ↔ 3.5G	0 ↔ 3.5G
PPC:	0 ↔ 1G	1G ↔ 4G	1G ↔ 4G	1G ↔ 4G
MIPS:	2G ↔ 4G	0 ↔ 2G	0 ↔ 2G	0 ↔ 2G
SH4:	2G ↔ 4G	0 ↔ 2G	0 ↔ 2G	0 ↔ 2G
ARMv4&5:	2G ↔ 4G	0 ↔ 32M	0 ↔ 32M	0 ↔ 32M
ARMv6&7:	2G ↔ 4G	0 ↔ 2G	0 ↔ 2G	0 ↔ 2G

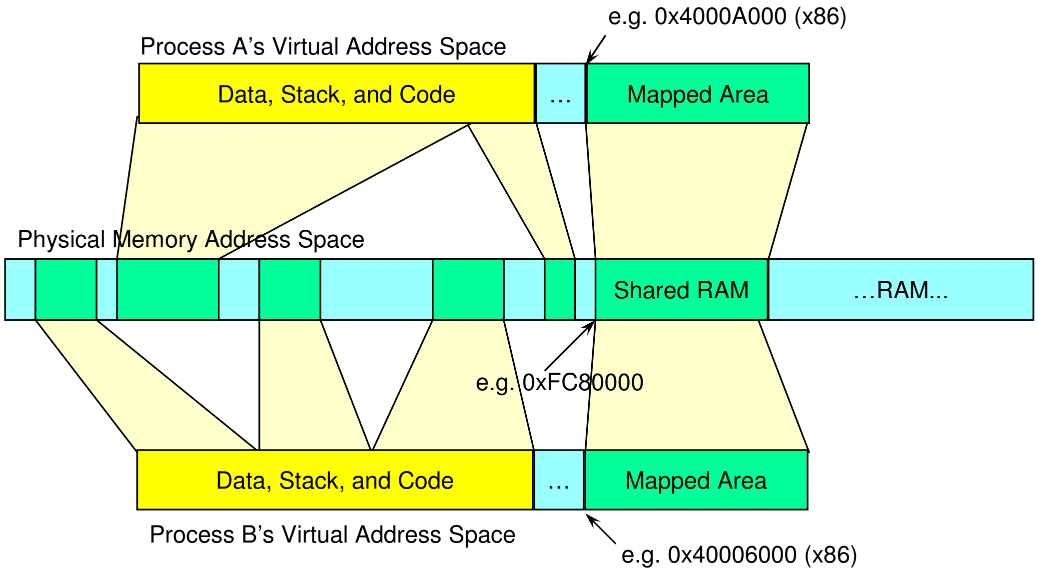
NOTES:

Using the standard armle procnto, you are restricted to 63 processes, rather than the usual 2048.

Memory Management - Shared Memory

QNX SOFTWARE SYSTEMS

Virtual addresses map to physical addresses:



Pointers that you deal with contain virtual addresses, not physical

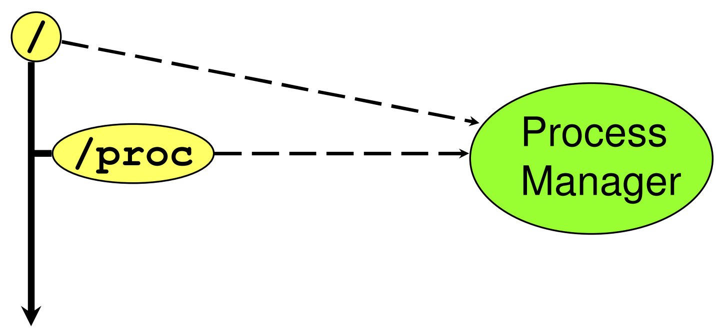
NOTES:

This can be viewed using the Memory Information View of the IDE.

At the commandline, you can use **`pidin mem | less`**

Process Manager - Pathname Management

When QNX Neutrino starts up, the entire pathname space is owned by **`procnto`**:



Any requests for file or device pathname resolution are handled by **`procnto`**.

NOTES:

These file spaces have the following meanings:

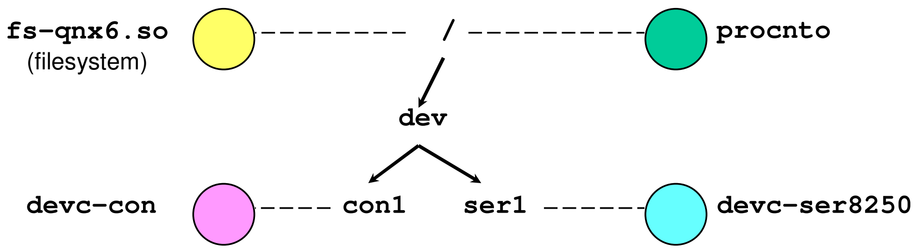
`/`

The root of the filesystem

`/proc`

The **`/proc`** filesystem. This is where the process manager places information about all of the currently executing processes

procnto allows resource managers to adopt a portion of the pathname space:



NOTES:

The power of this cannot be overstated. This mechanism allows various *resource managers* (which are a superset of the traditional UNIX device driver concept) to assume responsibility for a pathname prefix. This approach provides a consistent method of resolving pathnames to the actual resource managers that manage those pathnames.

Topics:

- Overview
- The Microkernel
- The Process Manager
- → Scheduling
- Adaptive Partitioning
- SMP
- Resource Managers
- System Library
- Shared Objects
- OS Services
- Conclusion

NOTES:

Threads have two basic states:

blocked

- waiting for something to happen
- there are lots of different blocked states depending on what they are waiting for, e.g.:
  - **REPLY** blocked is waiting for a IPC reply
  - **MUTEX** blocked is waiting for a mutex
  - **RECEIVE** blocked is waiting to get a message

ready

- capable of using the CPU
- two main ready states
  - **RUNNING** actually using the CPU
  - **READY** waiting while someone else is running

NOTES:

There are a couple more thread states that don't fall into these two categories.

DEAD - thread is dead, can not be recovered, never leaves this state

STOPPED - has been hit by a stop signal, will not continue processing until a continue signal is delivered

All threads have a priority:

- the priority range is 0 (low) to 255 (high)
- priority matters for ready threads only
- the kernel always picks the highest priority **`READY`** thread to be the one that actually uses the CPU (fully pre-emptive)
  - the thread's state becomes **`RUNNING`**
  - blocked threads don't even get considered
- most threads spend most of their time blocked
  - that is how CPU is shared between threads

NOTES:

A thread must have root privilege to set its priority above 63. (With the boundary, 63, configurable by a command-line option to procnto.)

For releases before 6.3.0, the priority range was 0 to 63.

Priorities:

Priorities

highest

```
(hardware interrupt handlers)
______________________________
255
 .
 .
21  devb-eide (one of many of
                 devb-eide's threads)
 .
 .
11
10  ksh
 9
 .
 .
 1
 0  procnto (idle thread)
```

Priority

lowest

NOTES:

Priorities range from 0 (lowest) to 255 (highest).

To examine the priorities of threads running on your target using the QNX IDE, use the Thread Information view in the System Information perspective. Look under the column, **Priority Name**.

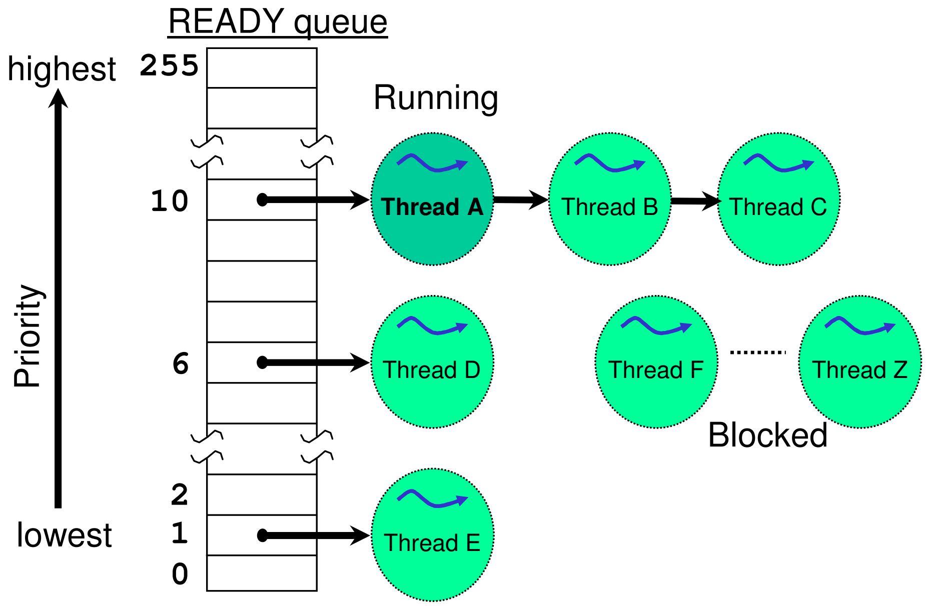
Using a shell, to examine the priorities of threads running on your processor do:

```
pidin
```

The priorities are shown in the `prio` column. The letters beside the numbers are the scheduling algorithms (we'll see this next).

Hardware interrupt handlers are not threads and are not scheduled along with other threads. However, hardware interrupt handlers are called as soon as the interrupt occurs, preempting any thread that is running.

The READY queue



NOTES:

The READY queue is a queue of which threads are currently RUNNING or READY (even though we call it the READY queue).

IMPORTANT: The highest priority READY thread is the one which gets the CPU (i.e. which runs). It then becomes the thread that is RUNNING. Lower priority READY threads will simply not get any CPU. Of course in the case of an SMP (multiprocessor system) a lower priority thread could run if there is a processor available.

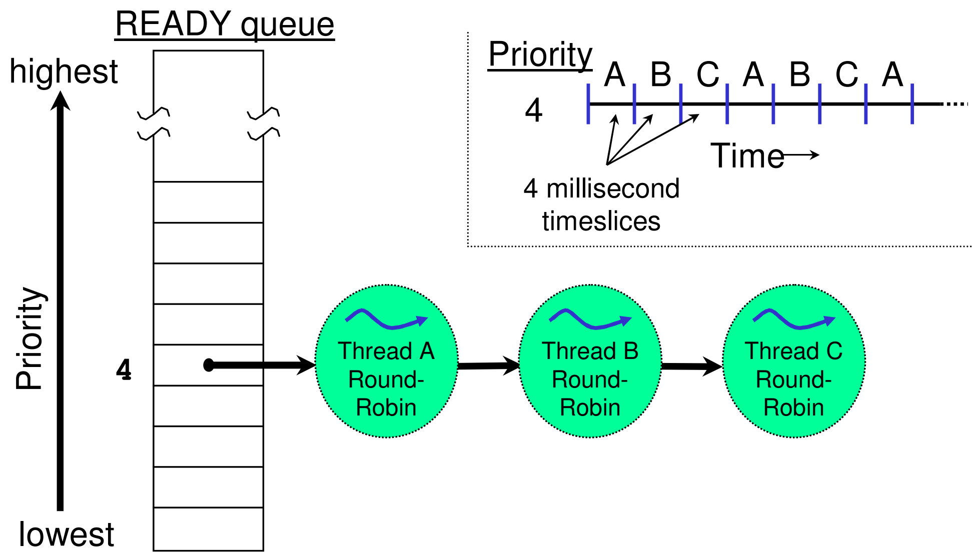
In the diagram above, Threads A, B and C are the highest priority READY threads. Since Thread A is next in the list at priority 10, A is the one that runs.

When a thread goes from being not READY (RECEIVE blocked, REPLY blocked, ...) to READY then it is put at the end of the list of READY threads for its priority.

Note that no change is made in the order of threads in the case of preemption by a higher priority thread -- as far as threads at the pre-empted priority are concerned, this is just an interruption, not a rescheduling event.

Scheduling algorithms- Round-robin

Scheduling algorithms: Round-robin



☞ There is an "other" algorithm that is currently the same as round-robin

QNX Neutrino Architecture 2010/06/21 R07

A subsidiary of Research In Motion Limited 45

NOTES:

In Round-robin a running thread continues to run until it:

- terminates
- voluntarily relinquishes control (blocks or yields the CPU or returns from a signal handler)
- is preempted by a higher priority thread
- consumes its timeslice

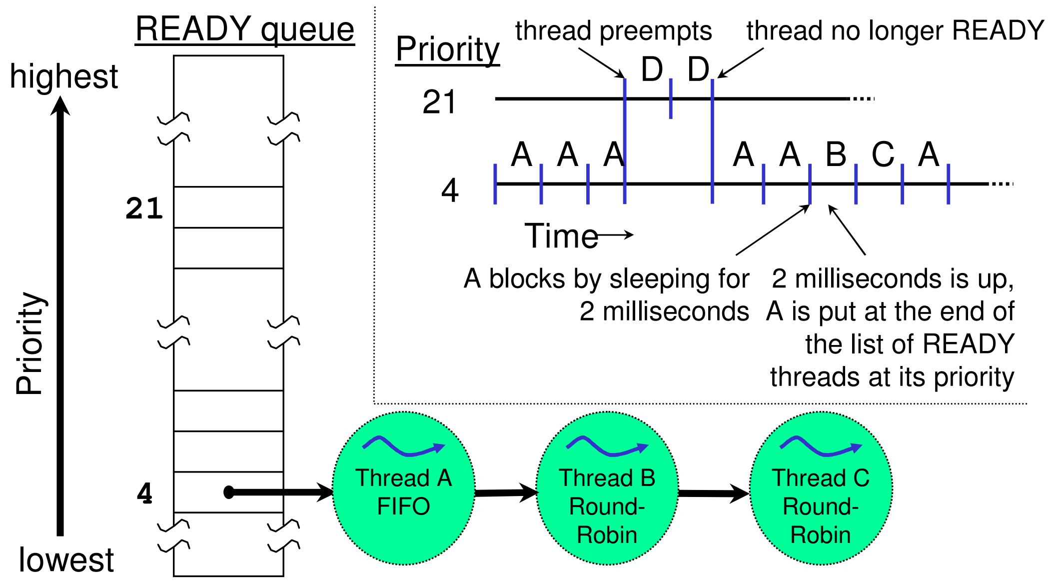
Its macro is **SCHED_RR** from **<sched.h>**

Round-robin threads are labelled by an **r** beside their priority in both the IDE and **pidin** output.

Threads labelled by an **o** are "Other" (**SCHED_OTHER**)scheduling algorithm. This is currently round-robin, but may change to an adaptive algorithm in the future.

With SMP, theads A, B, and C could be running at the same time on different processors.

Scheduling algorithms: FIFO



NOTES:

In FIFO a running thread continues to run until it:

- terminates
- voluntarily relinquishes control (blocks or yields the CPU or returns from a signal handler)
- is preempted by a higher priority thread (and when no higher priority threads are READY again, the preempted FIFOer continues.)

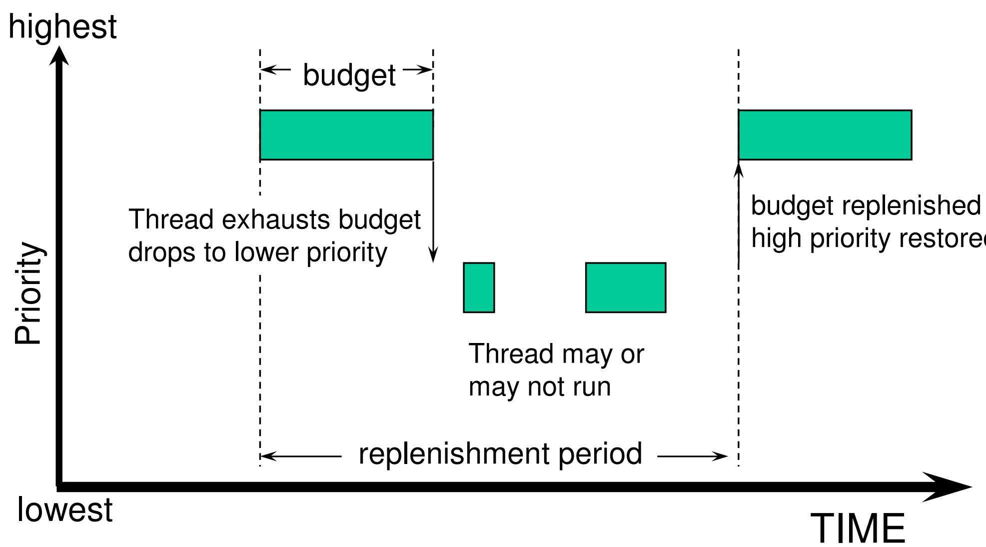
Its macro is **`SCHED_FIFO`** from **`<sched.h>`**

FIFO threads are labelled by an **`f`** beside their priority in both the IDE and **`pidin`** output.

Threads labelled by an **`o`** are "Other" scheduling algorithm. This is currently round-robin, but may change to an adaptive algorithm in the future.

With SMP, threads A, B, and C could be running at the same time on different processors.

Scheduling algorithms: Sporadic



NOTES:

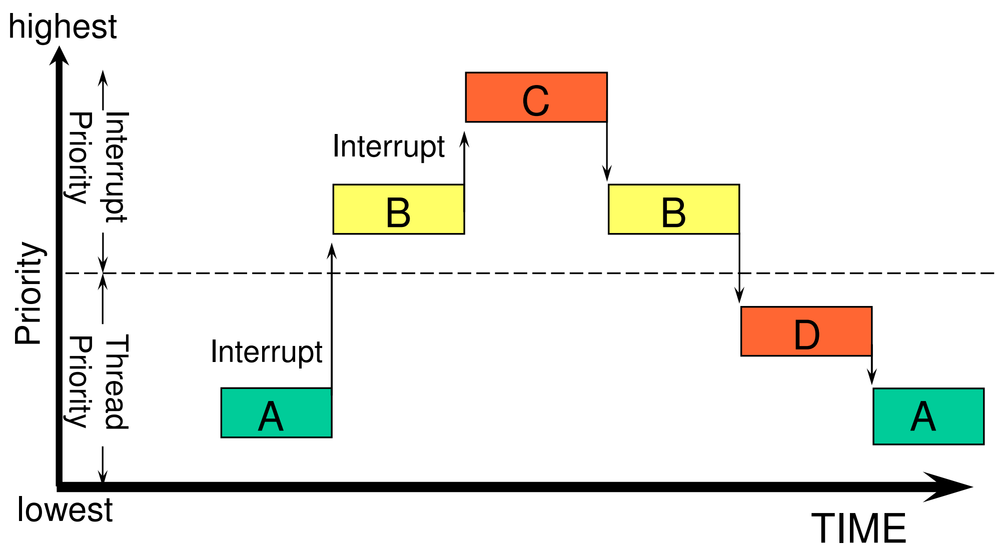
A thread is given a high and low priority, it will run at the high priority until its budget is exhausted, at which point its priority will drop to the low value. While at the low value, it may or may not run, depending on other threads in the system. After the replenishment period has elapsed, it will get a new budget and return to its high priority.

Its macro is **`SCHED_SPORADIC`** from **`<sched.h>`**

Sporadic threads are labelled by an **`s`** beside their priority in both the IDE and **`pidin`** output.

Threads labelled by an **`o`** are "Other" scheduling algorithm. This is currently round-robin, but may change to an adaptive algorithm in the future.

Interrupt Scheduling (preemptive):

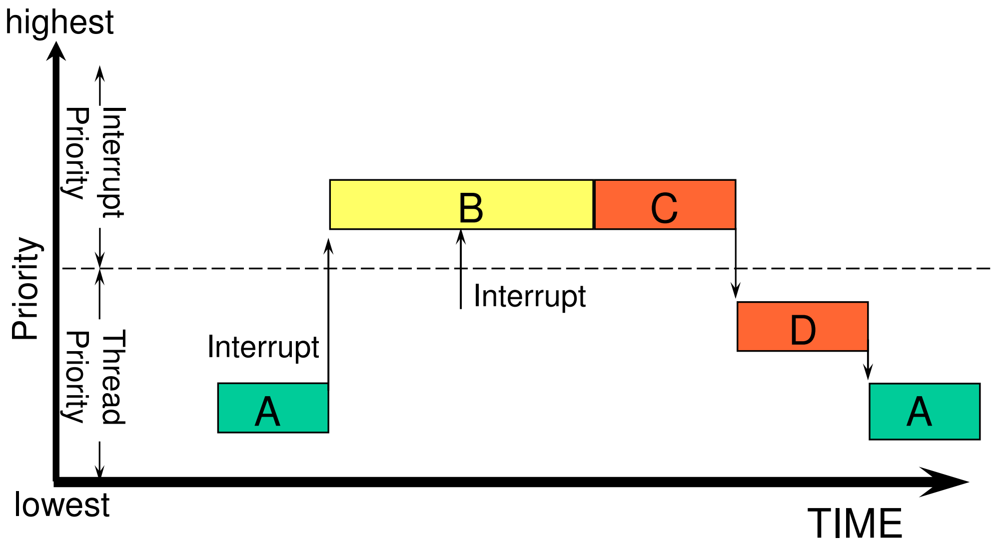


NOTES:

Thread A was running, and got interrupted by a hardware interrupt (B). That interrupt got interrupted by a higher priority interrupt (C). Handler C ran, and returned an event to make thread D READY. When C was done, B resumed running. When B was done, the highest priority thread ran, in this case D, because it was made READY by C.

On some target systems, the Programmable Interrupt Controller (PIC) handles the interrupt priorities. On some systems it is done by in software the interrupt_id and interrupt_eoi kernel callouts setup by the startup code. On other systems, there is no support for priority levels for interrupts.

Interrupt Scheduling (non-preemptive):



NOTES:

Thread A was running, and got interrupted by a hardware interrupt (B). While handler B was running, another hardware interrupt (C) was delayed until handler B completed. Handler C ran, and returned an event to make thread D READY. When all handlers had completed, the highest priority thread ran, in this case D, which was made READY by handler C.

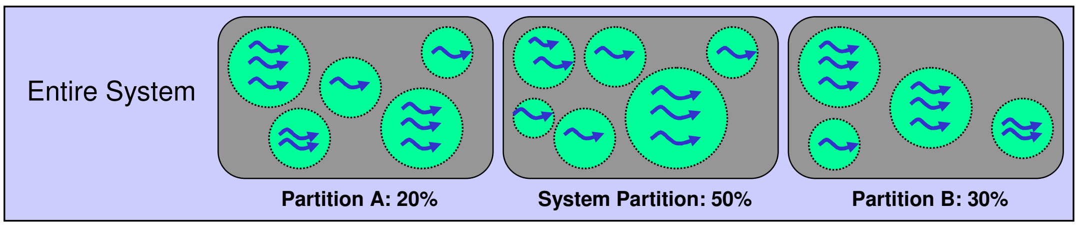
Topics:

- Overview
- The Microkernel
- The Process Manager
- Scheduling
- → Adaptive Partitioning
- SMP
- Resource Managers
- System Library
- Shared Objects
- OS Services
- Conclusion

NOTES:

System designer:

- creates scheduling partitions
- decides which partition processes/threads go into
  - child processes/threads go into parent's partition by default
- specifies minimum % CPU usage for each partition



NOTES:

Scheduling is ‘Adaptive’:

- if CPU time is not needed by a partition, it can go to another one
- if system is < 100% loaded:
  - scheduling works as it does without adaptive partitioning
  - CPU time goes to highest priority thread in system
- threads that have strict real-time requirements can be designated as being ‘critical threads’
  - e.g. interrupt handling threads
  - critical threads can borrow from future time if their partition is over budget and they need to run

NOTES:

Topics:

- Overview
- The Microkernel
- The Process Manager
- Scheduling
- Adaptive Partitioning
- → SMP
- Resource Managers
- System Library
- Shared Objects
- OS Services
- Conclusion

NOTES:

SMP:

- is short for Symmetrical MultiProcessor
- means that you are using a board that has more than one processor/CPU tightly coupled
- you don't have to write special code
- requires a different kernel:
  - e.g. **`procnto-smp`**, **`procnto-smp-instr`**, **`procnto-600-smp`**
- on an SMP system, threads of different priorities or multiple FIFO threads of the same priority may execute at the same time

NOTES:

Topics:

- Overview
- The Microkernel
- The Process Manager
- Scheduling
- Adaptive Partitioning
- SMP
- → Resource Managers
- System Library
- Shared Objects
- OS Services
- Conclusion

NOTES:

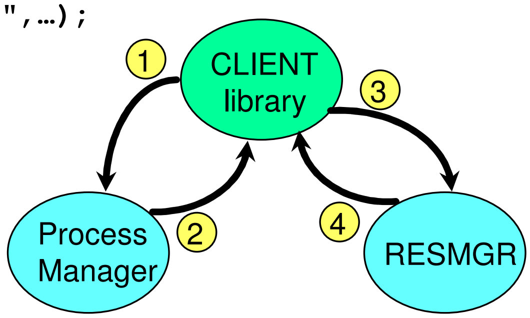
What is a resource manager?

- a program that looks like it is extending the operating system by:
  - creating and managing a name in the pathname space
  - providing a POSIX interface for clients (e.g. *open()*, *read()*, *write()*, ...)
- can be associated with hardware (such as a serial port, or disk drive)
- or can be a purely software entity (such as **`mqueue`**, the POSIX queue manager)

NOTES:

Interactions:

```
fd = open("/dev/ser1",…);
```



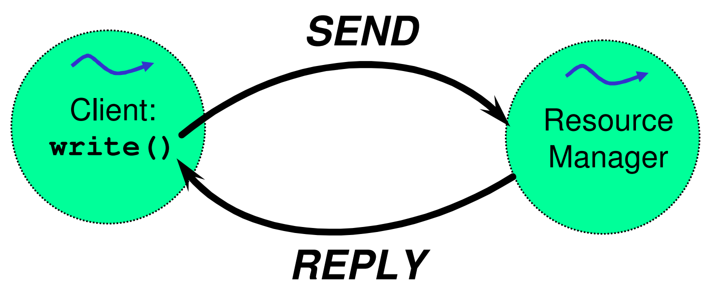
1. Client's library (*open()*) sends a "query" message
2. Process Manager replies with who is responsible
3. Client's library establishes a connection to the specified resource manager and sends an open message
4. Resource manager responds with status (pass/fail)

NOTES:

The application doesn't have to worry about these details -- it's all handled by *open()* in the C shared library.

How does *open()* find the process manager? Simple, it's a well known server.

Further communication is message passing directly to the resource manager:



NOTES:

Other notes:

- this setup allows for a lot of powerful solutions
  - debug "OS" drivers with a high-level (symbolic) debugger
  - distribute drivers across a QNX network
  - export access to your custom driver with a network file system such as NFS or CIFS
  - provide resiliency or redundancy of OS services
- QSS supplies a library that provides a lot of useful code to minimise the work needed to write one

NOTES:

By doing redirection with resource managers and with other methods, you can provide redundant versions of your own drivers, or even many OS services.

Topics:

- Overview
- The Microkernel
- The Process Manager
- Scheduling
- Adaptive Partitioning
- SMP
- Resource Managers
- → System Library
- Shared Objects
- OS Services
- Conclusion

NOTES:

Many standard functions in the library are built on kernel calls

- usually this is a thin layer, that may just change the format of arguments, e.g.
  - the POSIX function *timer_settime()* calls the kernel function *TimerSettime()*
    - it changes the time values from the POSIX seconds & nanoseconds to the kernel's 64-bit nanosecond representation
- we recommend using the standard calls
  - your code is more portable
  - you use calls that are going to be more familiar to and readable by your developers

NOTES:

But QNX is a microkernel

- so many routines that would be a kernel call, or have a dedicated kernel call in a traditional Unix become a message pass
- they build a message then call *MsgSend()* passing it to a server, e.g.
  - *read()* builds a message then sends it to a resource manager
  - *fork()* builds a message and sends it to the process manager

NOTES:

Still more functions supply an extra layer on top of something lower level

- e.g. the stdio functions provide local buffering on top of the underlying *read()* and *write()* calls so:
  - if you were wanting to read a byte at a time, *fread()* would be a good choice as it would locally buffer and only do the message pass every 1000 reads
  - if you were wanting to read 64k at a time, v*fread()* would break it down into 64 1K *read()*s, and you would be better off calling *read()* directly

NOTES:

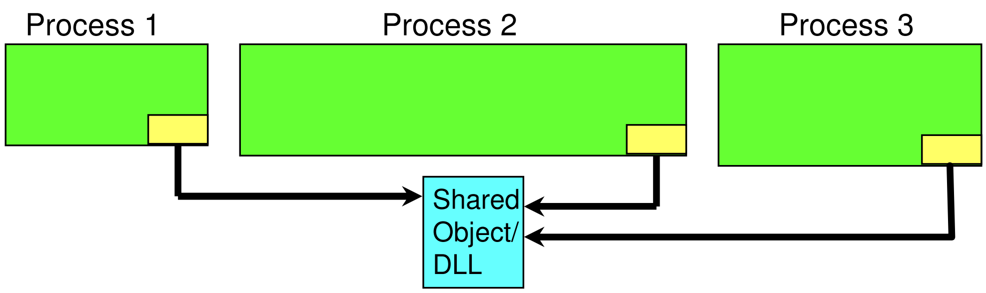
Topics:

- Overview
- The Microkernel
- The Process Manager
- Scheduling
- Adaptive Partitioning
- SMP
- Resource Managers
- System Library
- → Shared Objects
- OS Services
- Conclusion

NOTES:

Shared Objects:

- are libraries loaded and linked at run time
- one copy used (shared) by all programs using library
- also sometimes called DLLs
  - shared objects and DLLs use the same architecture to solve different problems



NOTES:

The term DLL is often used to refer to shared objects that are loaded explicitly by processes by calling *dlopen()* once the processes are already up and running.

Topics:

NOTES:

System Services

QNX is a microkernel:

- most system services are delivered by a process
- if you want the service, you run the process
  - if you don't want/need the service, you don't pay the code and data overhead for the service
- services can be dynamically configured/removed as needed

NOTES:

System Services

Some of the service/processes are:

`pps`	Persistent Publish/Subscribe IPC
`mqueue/mq`	POSIX message queues IPC
`dumper`	Core dump creation
`pipe`	Unix pipes
`devb-*`	Filesystems, usually rotating media
`devf-*`	Filesystems, NOR flash
`io-pkt-*`	Networking access
`slogger`	QNX system logger
`syslogd`	Unix syslog support
`pci-*`	PCI bus access and configuration

NOTES:

Topics:

NOTES:

Conclusion

QNX SOFTWARE SYSTEMS

You learned that:

- QNX Neutrino is a microkernel architecture OS
- most OS services are delivered by cooperating processes
- processes own resources and threads run code
- QNX Neutrino does pre-emptive scheduling
  - only **`READY`** threads are schedulable, blocked threads are not

NOTES: