Multicore

NOTES:

Topics:

→ **Overview**

Scheduling

Synchronization

Conclusion

NOTES:

Overview

SMP:

- is short for Symmetrical MultiProcessing
  - now generally called multicore
  - still used in naming the kernel
- means a system that has more than one processor/CPU tightly coupled
  - independent processors
  - shared hardware, RAM, bus, etc
- multicore processors, where the processors share one chip, are a common modern case of this

NOTES:

Multicore and SMP aren't exactly the same thing, but instead are heavily overlapping concepts with small differences in which edge cases fit in.

Overview

To use multicore, nothing needs to be done:

- the kernel is already multicore aware as of SDP7.0
  - you don't have to write any special code
- **`procnto-smp-instr`** is designed to support single and multi-core processors
  - if run on a single-core processor, will run in single-core mode
  - on a multi-core processor, will run in multi-core mode
  - the BSP must support multi-core as well

NOTES:

You don't run **`procnto-smp-instr`** from the command line. You must build it into your boot image.

Topics:

- Overview
- → Scheduling
- Synchronization
- Conclusion

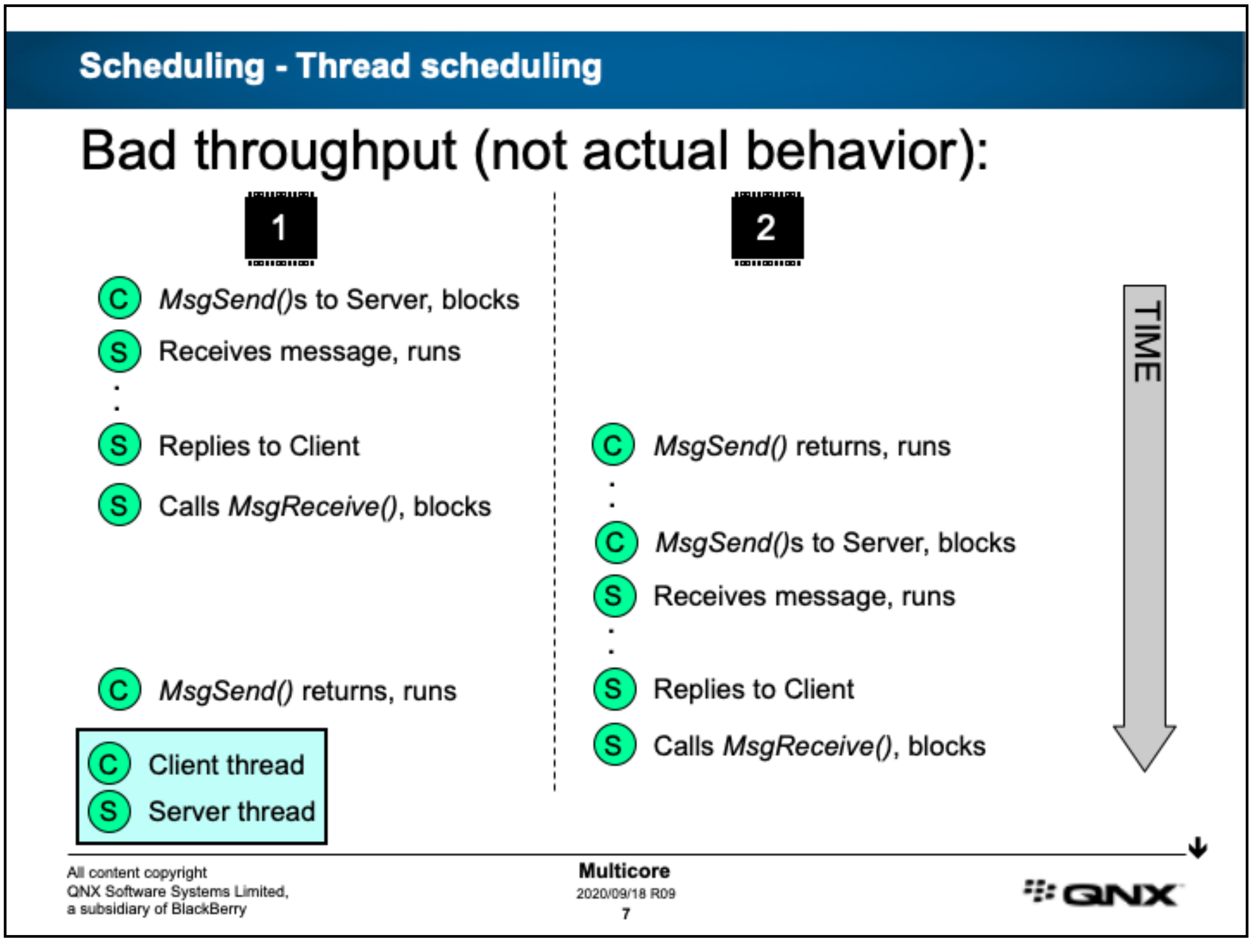
NOTES:

With the multicore scheduler:

- when a thread becomes runnable, usually:
  - we find the core with the lowest priority running thread
  - if the newly runnable thread is higher priority than that thread, run the new thread on that core
  - this behavior can be configured with:

  `SchedCtl( SCHED_CONFIGURE, … );`
- we guarantee that the highest priority thread that is ready to use the CPU is running
  - this is the same thread that would be scheduled uniprocessor
- but, sometimes lower priority thread(s) may be running while higher priority thread(s) are ready
  - a trade-off between latency and throughput…

NOTES:

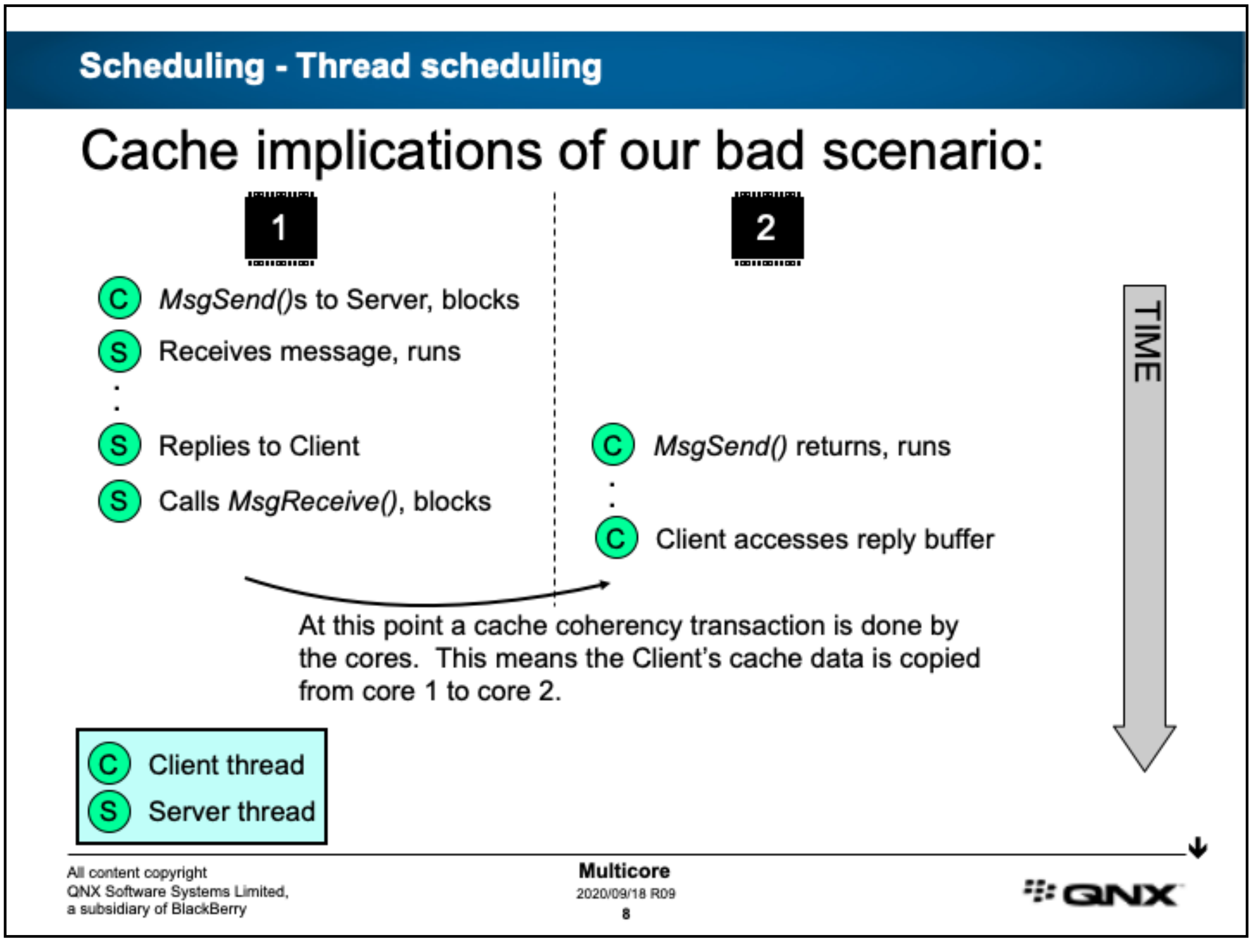


NOTES:

In the above scenario there are two threads: a Client thread and a Server thread. The Client is sending messages to the Server which receives the messages and replies back.

Notice that the threads keep changing CPUs. When the Server replies to the Client, the Server is still ready on CPU 1 so the Client runs on CPU 2 instead. The Server then goes back to the top of its receive loop and blocks again to wait for the next message. The Client, now on CPU 2, sends another message to the Server causing the Server to run on CPU 2. When the Server replies to the client, the Server is still ready on CPU 2 so the Client runs on CPU 1 instead...

This is called thrashing. The threads keep moving from one CPU to another. As the next slide illustrates, this is bad because it causes cache coherency operations.



NOTES:

This design causes a lot of cache coherency transactions as the CPUs have to keep copying their cache data in order to follow where the threads are now running. Remember, this is not how it behaves with the default configuration. This is just meant to illustrate the cost to throughput.

Scheduling - Thread scheduling

Actual behavior:

1

- C *MsgSend()*s to Server, blocks
- S Receives message, runs
- .
- .
- S Replies to Client
- S Calls *MsgReceive()*, blocks
- C *MsgSend()* returns, runs
- .
- .

Once a server has replied to a client it typically goes back to a *MsgReceive()* to wait for another message.
So even though the Client becomes ready here...
... we don't schedule the client until…
The server blocks. Then we run the Client.

The end result is that:
- they don't thrash from one core to another and
- cache coherency transactions aren't needed.

TIME

C Client thread
S Server thread

QNX

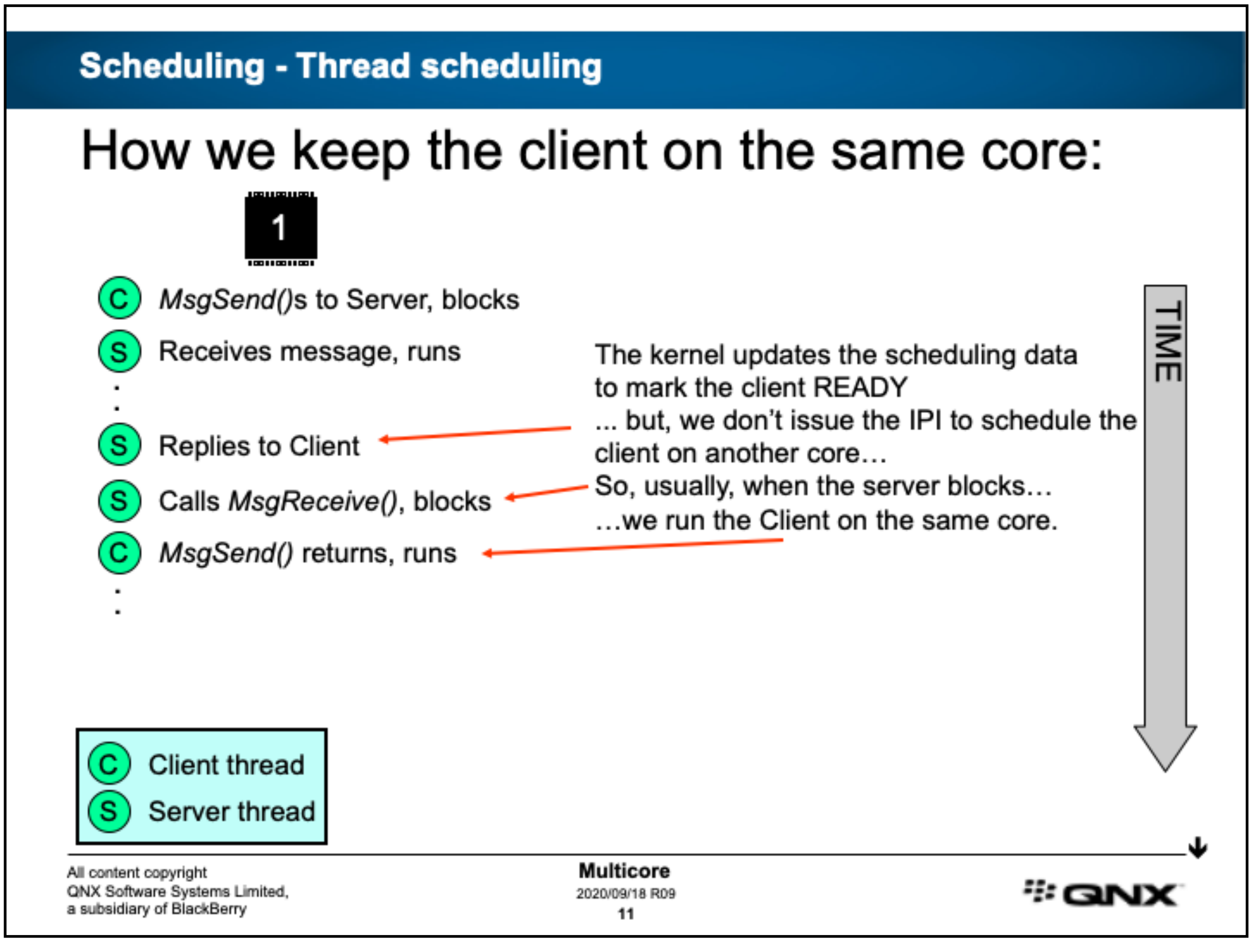
NOTES:

Multicore Scheduling

How multicore scheduling is done in QNX:

- the scheduling state is stored in memory
  - the data is shared by all cores
- when a thread change state from RUNNING on a core
  - the kernel runs another thread on that core
- when a thread changes state to a runnable state on a particular core:
  - the kernel, running on that core, calculates a new scheduling state
  - if it needs to run a different thread on the current core, it does so
  - if a different thread needs to be scheduled on another core, it issues an InterProcessor Interrupt (IPI) to that core and exits the kernel
- the other core enters the kernel due to the IPI
  - looks at the scheduling data, and schedules a (new) thread
  - updates the scheduling state data
  - perhaps issues another IPI

NOTES:



NOTES:

"usually" because a number of things may interfere: a higher priority thread may get scheduled on the core before either client or server runs, or the kernel may end up running the scheduling code on another core before the server blocks on this core, and that may result in the client migrating.

SchedCtl – SCHED_CONFIGURE

`SchedCtl(SCHED_CONFIGURE, …)` takes a structure with two parameters:

- **`low_latency_priority`**
  - if a thread becomes runnable and is
    - higher priority than the current thread on the current core
    - higher priority than this setting
  - it preempts on the current core immediately
- **`migrate_priority`**
  - if a thread
    - gets preempted on this core
    - is higher than this priority
    - there is a lower-priority thread running on another core
  - then migrate it immediately to another core
- both default to **`INT_MAX`**

NOTES:

A thread scheduling optimization:

- normally the kernel puts a newly scheduled thread on the core with the lowest priority thread
- if there are several such cores available, then the kernel tries to run that thread on the same core that it last ran on

In this example, let's pretend that when Thread D last ran it was on core 4. If Thread D becomes ready to run now with the cores in use as in this diagram, the kernel will run it on core 4.

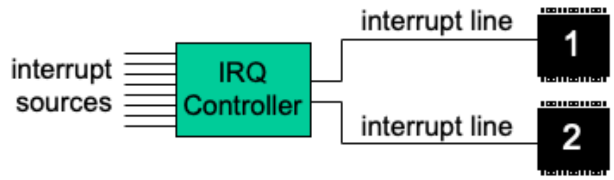
NOTES:

The kernel will try to run the thread on the same core that it last ran on because it is possible that some of the thread's code and data are still in that CPU's cache.

This is often called "soft affinity".

Where do interrupt handlers run?

– controlled by a mapping of interrupt sources to cores
  - for some controllers the **`startup`** code can set it up, for others it cannot be changed



– interrupt handlers run on the core that gets the interrupt
– if interrupt handling causes an event to be delivered then the recipient thread will typically want to use the same core
  - can be controlled using processor affinity...

NOTES:

Scheduling - Processor affinity

Processor affinity:

- you can arrange it so that the kernel will run a thread only on a certain core or set of cores
- to do this, use the *ThreadCtl()* function:

```
ThreadCtl(_NTO_TCTL_RUNMASK, (void *) runmask);
```

where each bit in **runmask** represents a core

bits: **1001**

core 4   core 1

Example:

```
int runmask;
runmask = 0x9; /* 0x9 = 1001, so run core 1 or 4*/
ThreadCtl(_NTO_TCTL_RUNMASK, (void *)runmask);
```


QNX

NOTES:

By default, processor affinity is not inherited by children. In order to have child threads inherit the same affinity as the parent, you would need to use:

```
ThreadCtl(_NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT, …)
```

See documentation on *ThreadCtl()* for further details.

Why use processor affinity?

- you can keep non-realtime threads on a specific core
  - in general this is not necessary since the highest priority ready thread will always pre-empt a lower priority thread
- to make better use of the CPU cache?
  - most useful after interrupt handling
  - chances are that some of the thread's code/data is in a core's cache
  - soft affinity is usually enough

NOTES:

Where does the kernel run?

- to answer this you need to know when the kernel runs, it runs when:
  - a hardware interrupt is generated
    - we already saw how interrupts are assigned to cores
    - timeslicing of threads is done by the kernel as a part of handling the timer interrupt
  - a fault or exception occurs (e.g. a bus error, segment violation)
    - kernel handles it on the core that the fault occurred on
  - a kernel call is made
    - kernel runs on the same core as the caller

NOTES:

Multicore

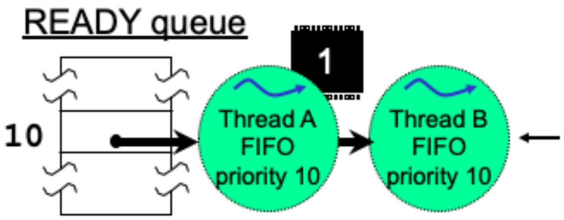
Topics:

- Overview
- Scheduling
- → **Synchronization**
- Conclusion

NOTES:

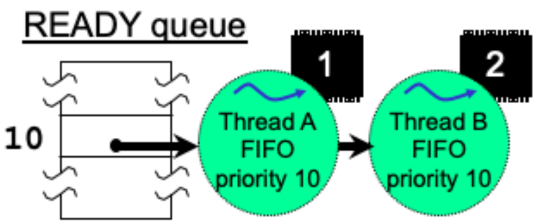
Synchronizing between threads:

– on a uniprocessor system synchronization can sometimes be done using FIFO scheduling



Thread A will keep the CPU at priority 10 until it blocks, yields the CPU or returns from a signal handler. Only then would Thread B get the CPU.

– on an multicore system they may run at the same time



So this type of synchronization cannot be used on multicore systems.

NOTES:

Synchronization - Between threads

Standard thread synchronization functions:

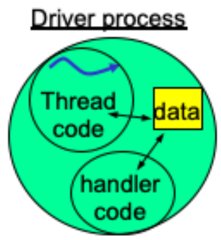
- barriers, mutexes, condvars, semaphores and all of their derivatives can still be used for synchronization on multicore
  - we guarantee that all the thread sync functions will work as expected, providing proper behaviour in multicore systems

NOTES:

Synchronization - Between thread and interrupt handler

Synchronizing between a thread and an interrupt handler:

- sometimes a thread and an interrupt handler within a driver process will have to share data



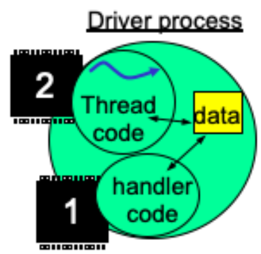
- using an event to schedule a thread completely avoids this problem
  - *InterruptAttachEvent()* rather than *InterruptAttach()*

NOTES:

Synchronization - Between thread and interrupt handler

The problem:

- on a uniprocessor system:
  - an interrupt handler can pre-empt a thread at anytime
    - this is because interrupt handlers have higher priority than any thread
  - a thread cannot pre-empt an interrupt handler
- on an multicore system:
  - the interrupt handler and thread can both be running at the same time!



NOTES:

Synchronization - Between thread and interrupt handler

Solution 1 of 2 - use the atomic*() functions:

`atomic_add`	does += some value
`atomic_add_value`	*atomic_add()*, and returns original value
`atomic_clr`	does &= ~some value
`atomic_clr_value`	*atomic_clr()*, and returns original value
`atomic_set`	does |= some value
`atomic_set_value`	*atomic_set()*, and returns original value
`atomic_sub`	does -= some value
`atomic_sub_value`	*atomic_sub()*, and returns original value
`atomic_toggle`	does ^= some value
`atomic_toggle_value`	*atomic_toggle()*, and returns original value

These functions:

- are guaranteed to complete correctly despite pre-emption or interruption, and between cores
- can be used between threads
- can be used between threads and an interrupt handler

NOTES:

Synchronization - Between thread and interrupt handler

Solution 2 of 2 - use an exclusion lock:

- declare a spinlock variable such that both your thread and interrupt handler can access it:

```
intrspin_t spinlock;
```

  - it must initially contain zeros
  - if it is not initialized that way, use *memset()* to zero it
- then, in both your thread and your interrupt handler do the following to access the data:

```
InterruptLock(&spinlock);
/* access your data */
InterruptUnlock(&spinlock);
```

☞ you typically keep the time between the above lines as short as possible

☞ you must have called *ThreadCtl(_NTO_TCTL_IO, 0)* for these functions to work

QNX

NOTES:

InterruptLock() disables interrupts and spins, waiting for the spinlock variable to have a certain value.

InterruptUnlock() sets the spinlock value to what *InterruptLock()* is waiting for and then enables interrupts.

Synchronization - Between thread and interrupt handler

InterruptDisable()/InterruptEnable():

- always available
- only disables on the calling core
- will not provide the expected protection with multicore

Always use *InterruptLock()*/*InterruptUnlock()*

- even on a uniprocessor system
  - driver, when moved to multicore, will work
  - overhead is small enough to not matter

NOTES:

Multicore

Topics:

- Overview
- Scheduling
- Synchronization
- → **Conclusion**

NOTES:

Conclusion

You learned:

- how the multicore scheduler works
  - and how to tune its behavior
- how and why to use processor affinity
- how to synchronize between threads and interrupt handlers on multicore

NOTES: