QNX® Neutrino® Architecture

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

Topics:

→ **Overview**

The Microkernel

The Process Manager

Scheduling

Resource Managers

System Library

Shared Objects

OS Services

Security

Boot Sequence

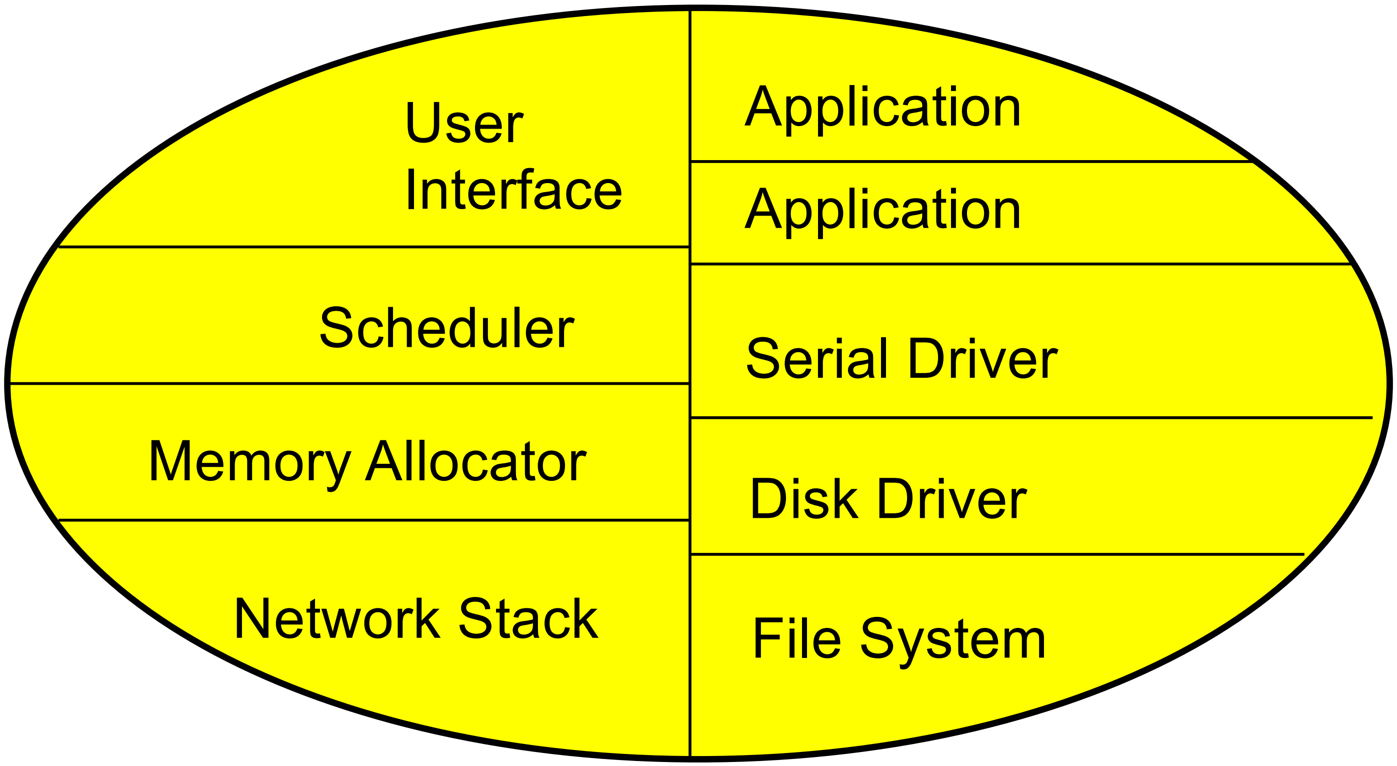
Conclusion

Portability

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

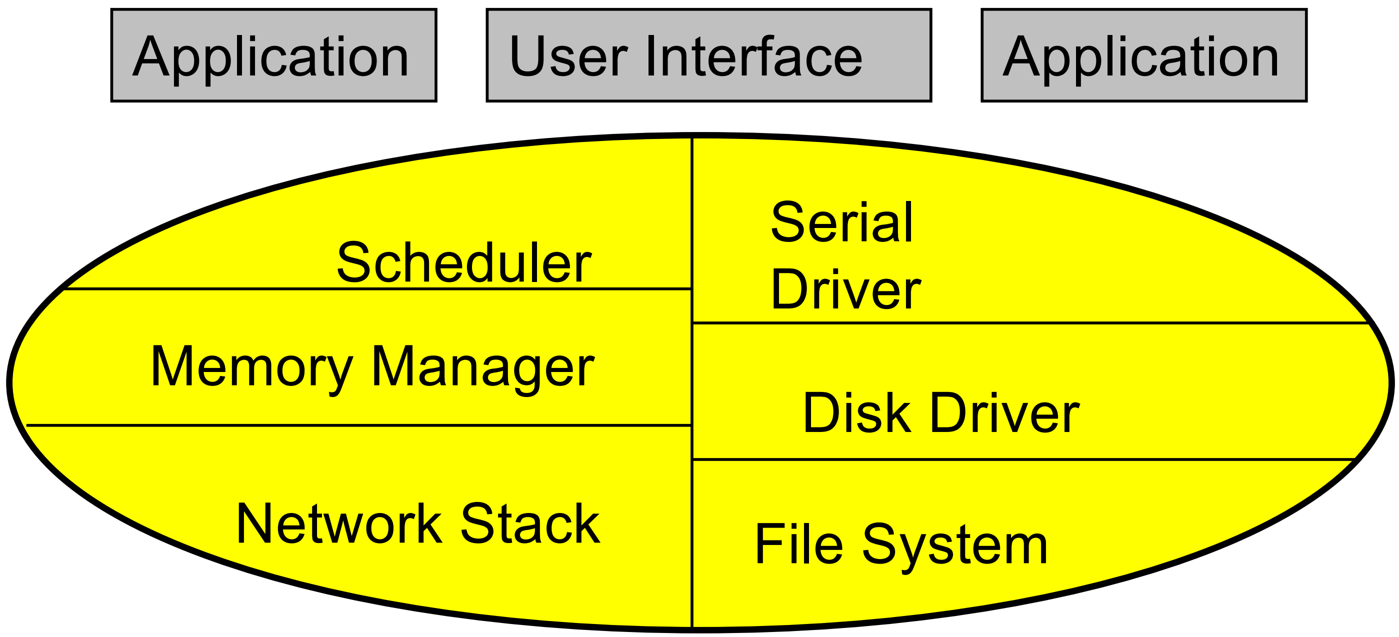
- POSIX.1
  - Unix, threads, timers, signals, etc
- ANSI C/C++
  - GNU Compiler Chain

In a traditional Real-Time Executive:



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

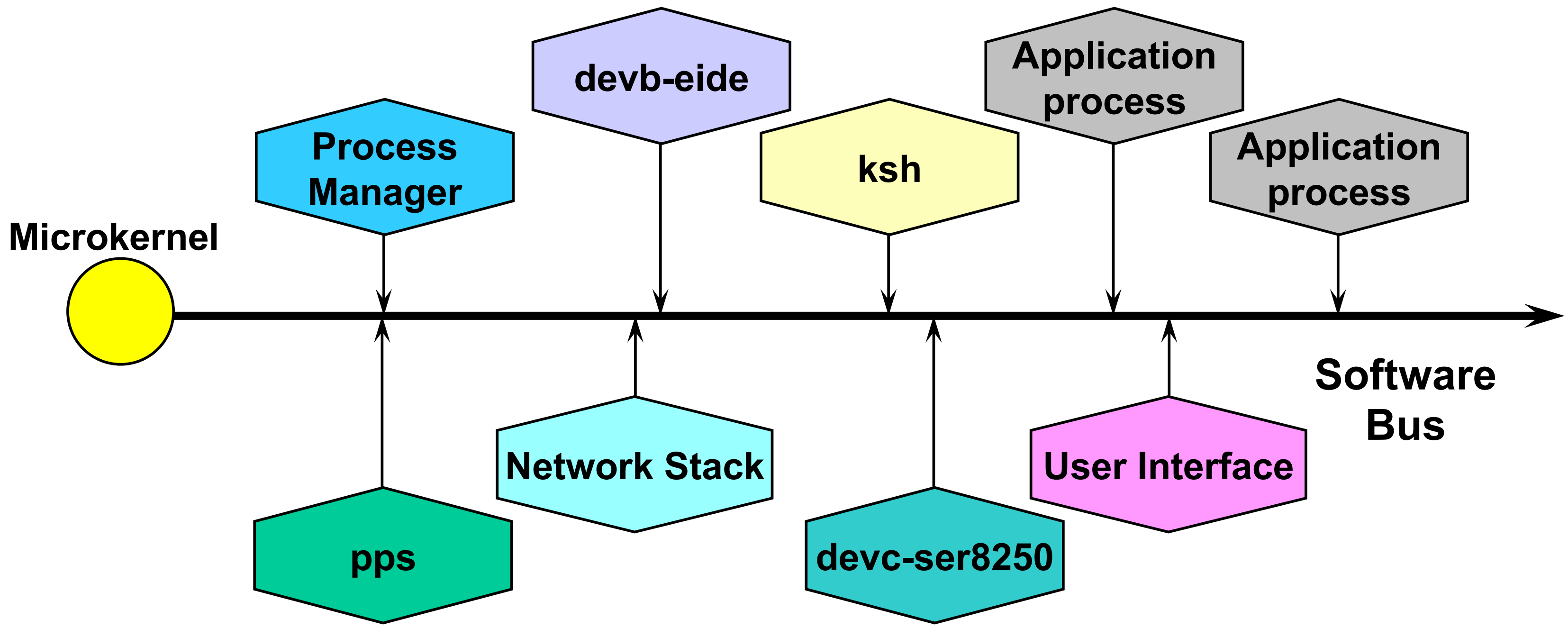
In a traditional Monolithic kernel OS:



Monolithic Kernel

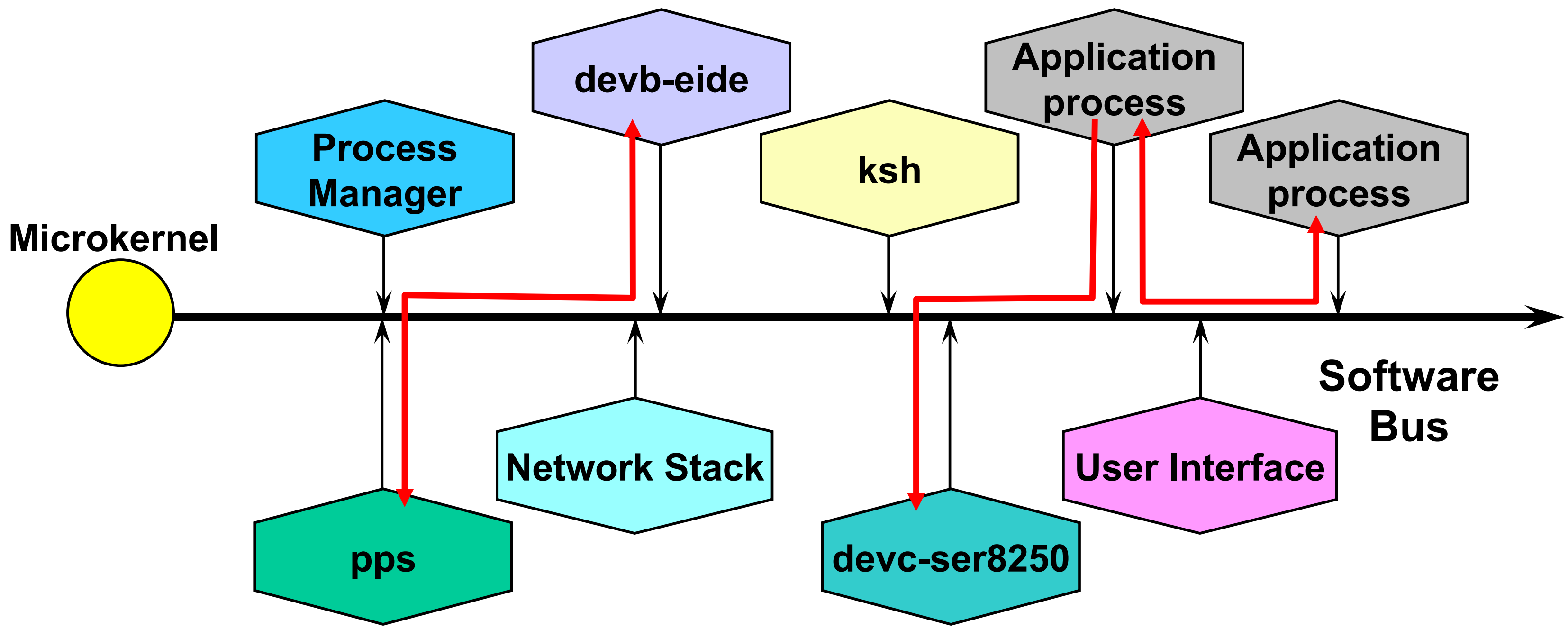
- The kernel contains the OS kernel functionality and all drivers, so driver development is complex and debugging can be painful.
- Applications are processes in protected memory space, so the kernel is protected from applications and applications are protected from each other.

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.

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

Examples of processes are:

- Disk Drivers
  - devb-eide, devb-virtio
- Network Stack
  - io-pkt
- Character Drivers
  - devc-ser8250, devc-con
- GUI components
  - screen
- Bus managers
  - pci-server, io-usb-otg
- System daemons
  - cron, inetd, mqueue, qconn

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

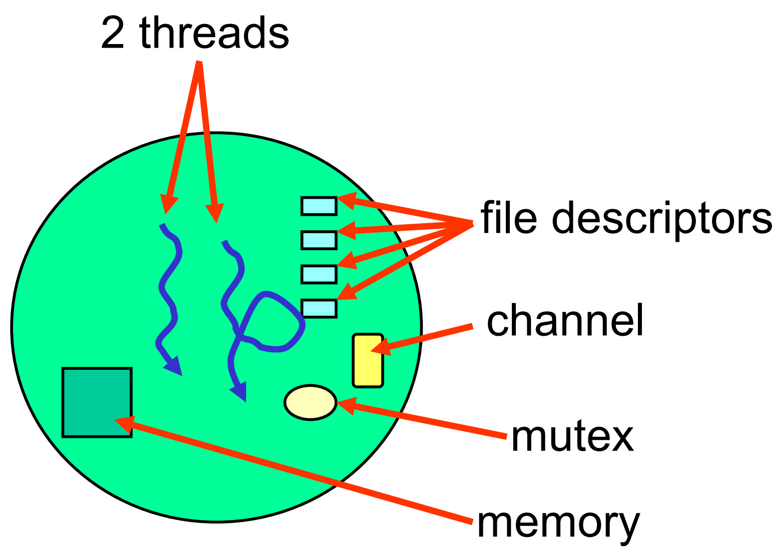
What is a process?

– 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



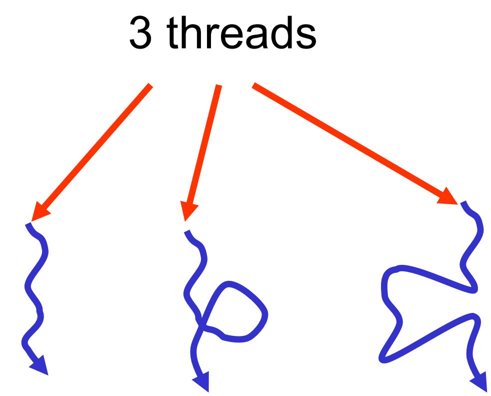
Resources owned by one process are protected from other processes

What is a thread?

– a thread is a single flow of execution or control

– a thread has some attributes:

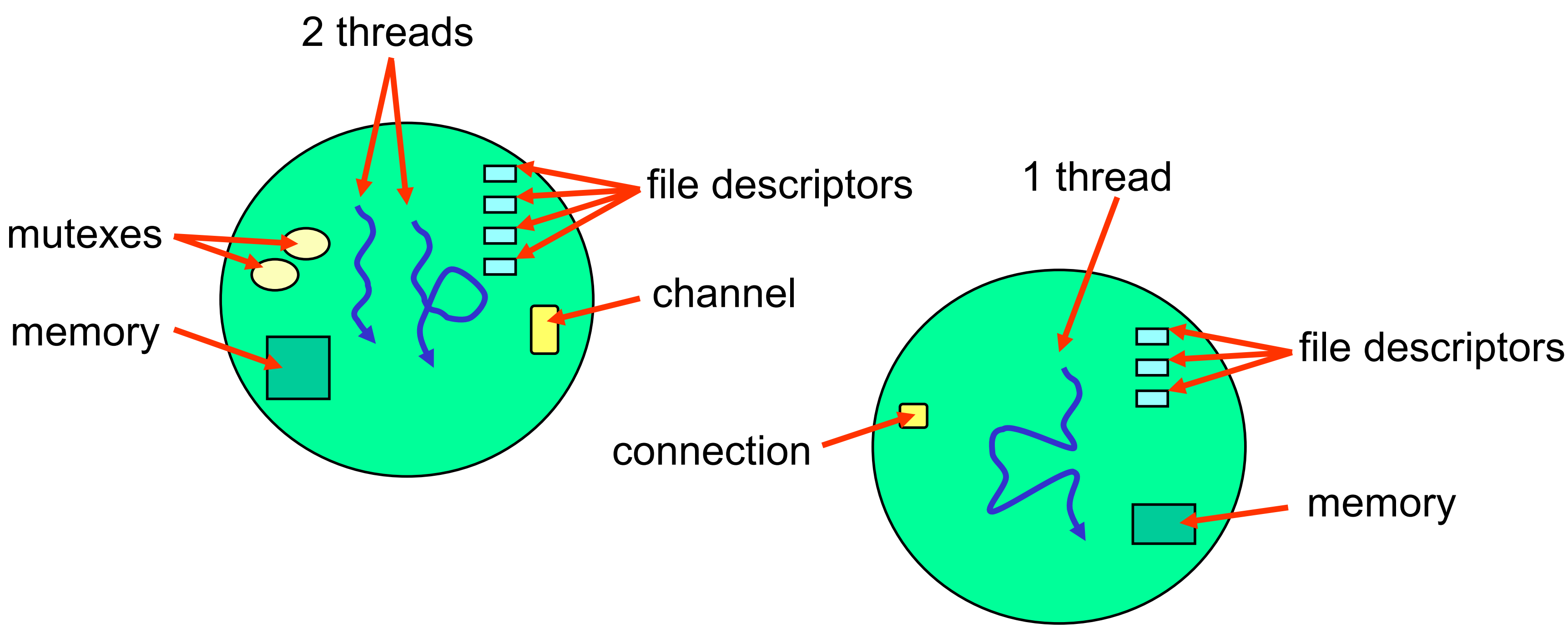
- priority
- scheduling algorithm
- register set
- CPU mask for multicore
- signal mask
- and others



– all its attributes have to do with running code

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

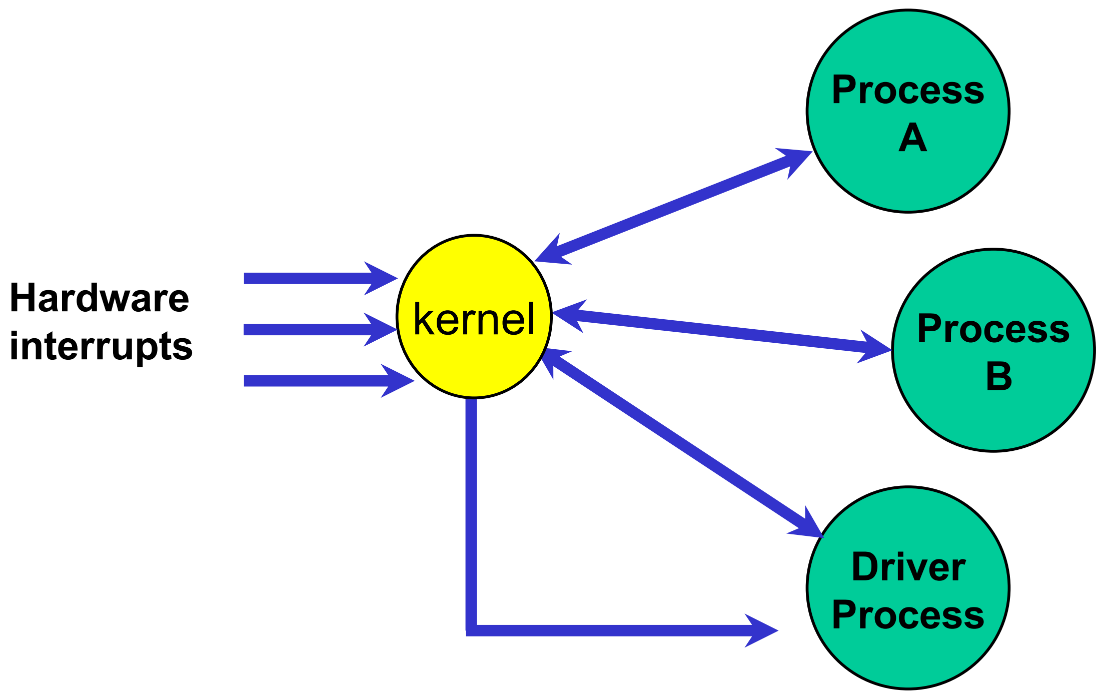
Processes and threads:

- 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

Topics:

- Overview
- → The Microkernel
- The Process Manager
- Scheduling
- Resource Managers
- System Library
- Shared Objects
- OS Services
- Security
- Boot Sequence
- Conclusion

The kernel is the core of your system:



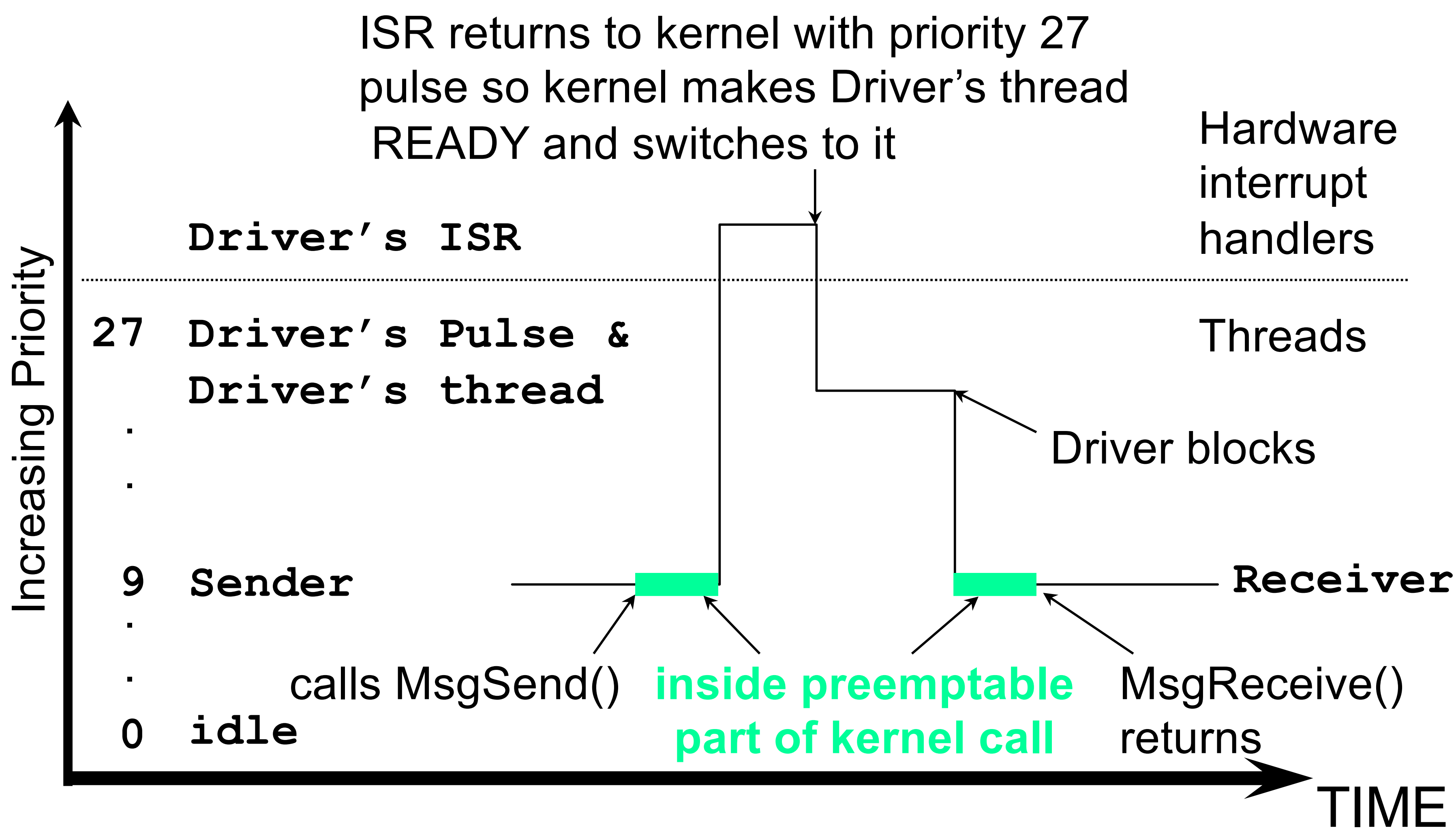
The kernel is special:

- 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
  - in the QNX C library these are recognizable as they are in CamelCase
  - e.g. *MsgSend()*, *ThreadCreate()*, *TimerCreate()*
- most of the other sub-systems, including user applications, communicate with each other using the message passing provided by the kernel through kernel calls

Possible implication of making kernel calls:

- you'll be executing code in the kernel for the duration of the call
- what if a time critical event occurs?
  - will handling it have to wait until the kernel call is done?
  - kernel call preemption helps with this...

Kernel calls are preemptable:

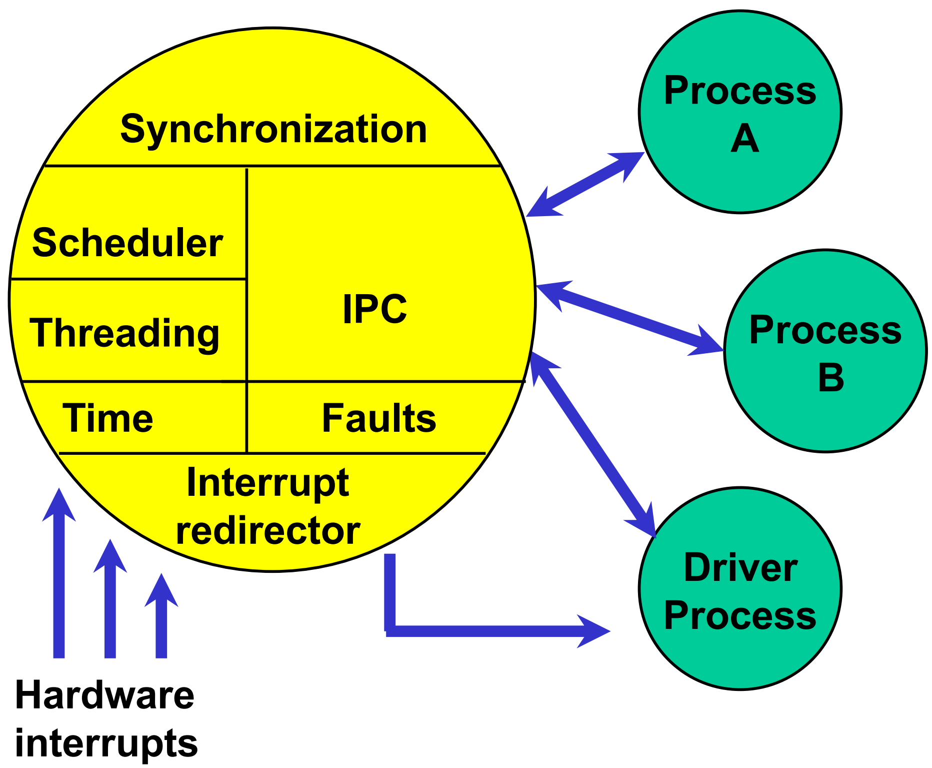


So what does this mean?

More trade-offs:

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

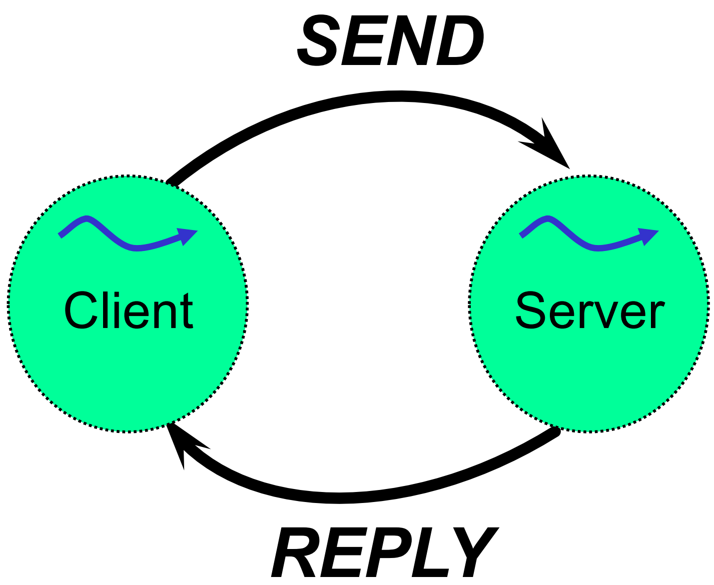
The kernel provides:



The forms of IPC provided by the kernel:

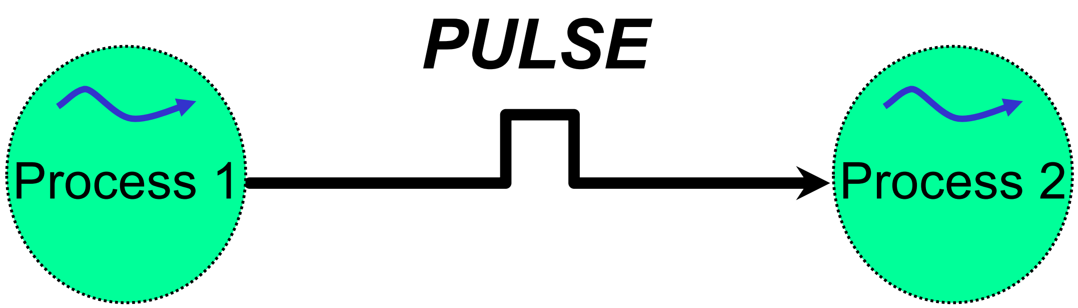
- Messages
  - exchanging information between proceses
- Pulses
  - delivering notification to a process
- Signals
  - interrupting a process and making it do something different (usually termination)

Native QNX Neutrino Messages:



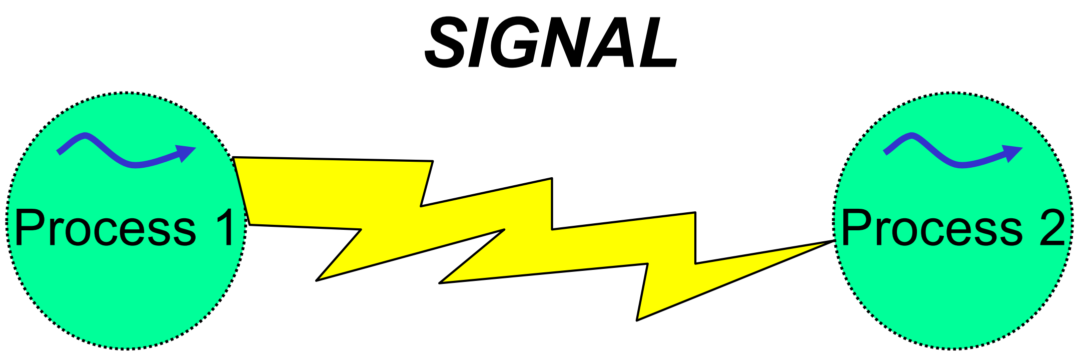
Native QNX Neutrino Pulses:

- used for event notification: “something happened”



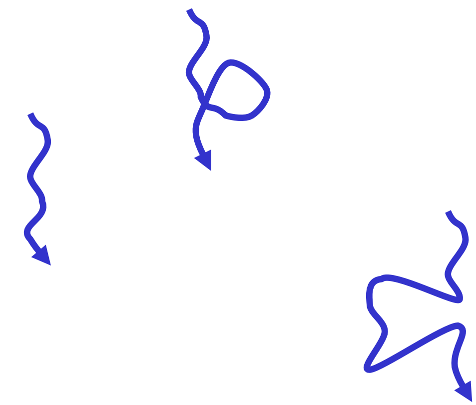
POSIX Signals:

– interrupt another process



Thread Functions:

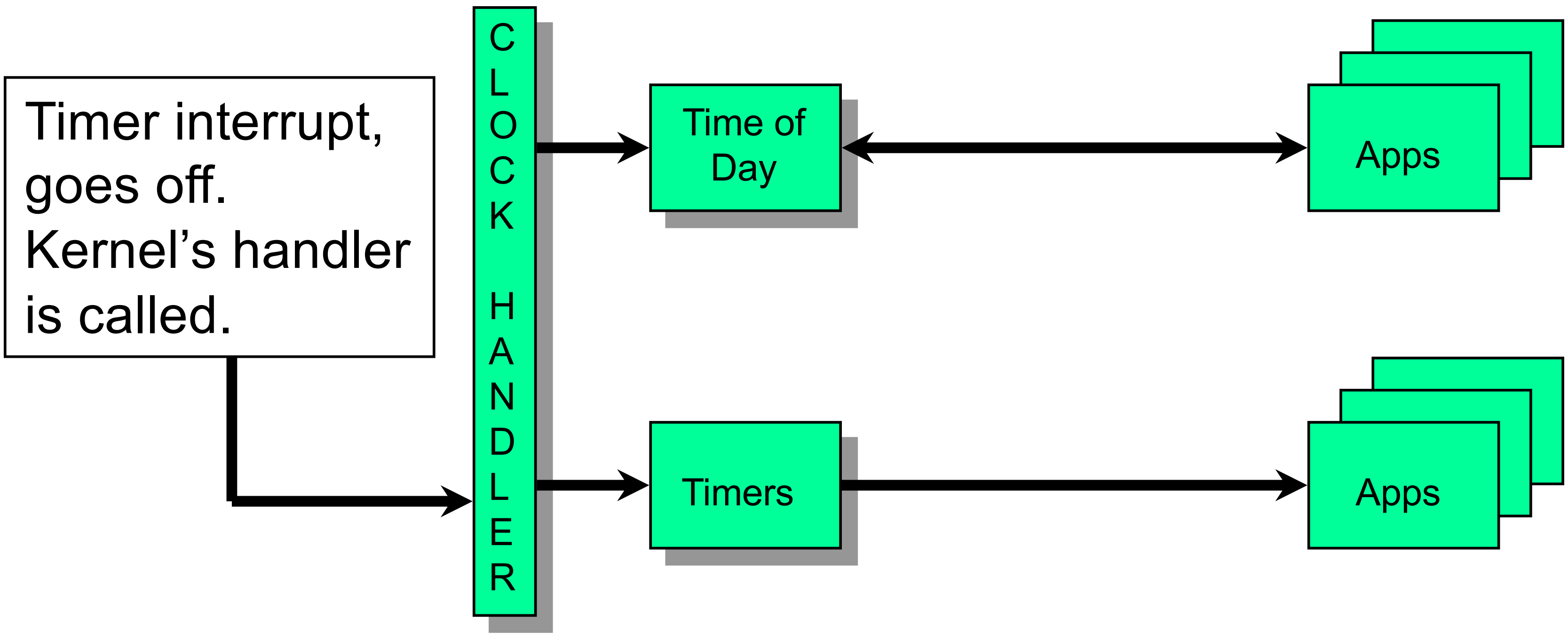
- create / terminate threads
- wait for thread completion
- change thread attributes



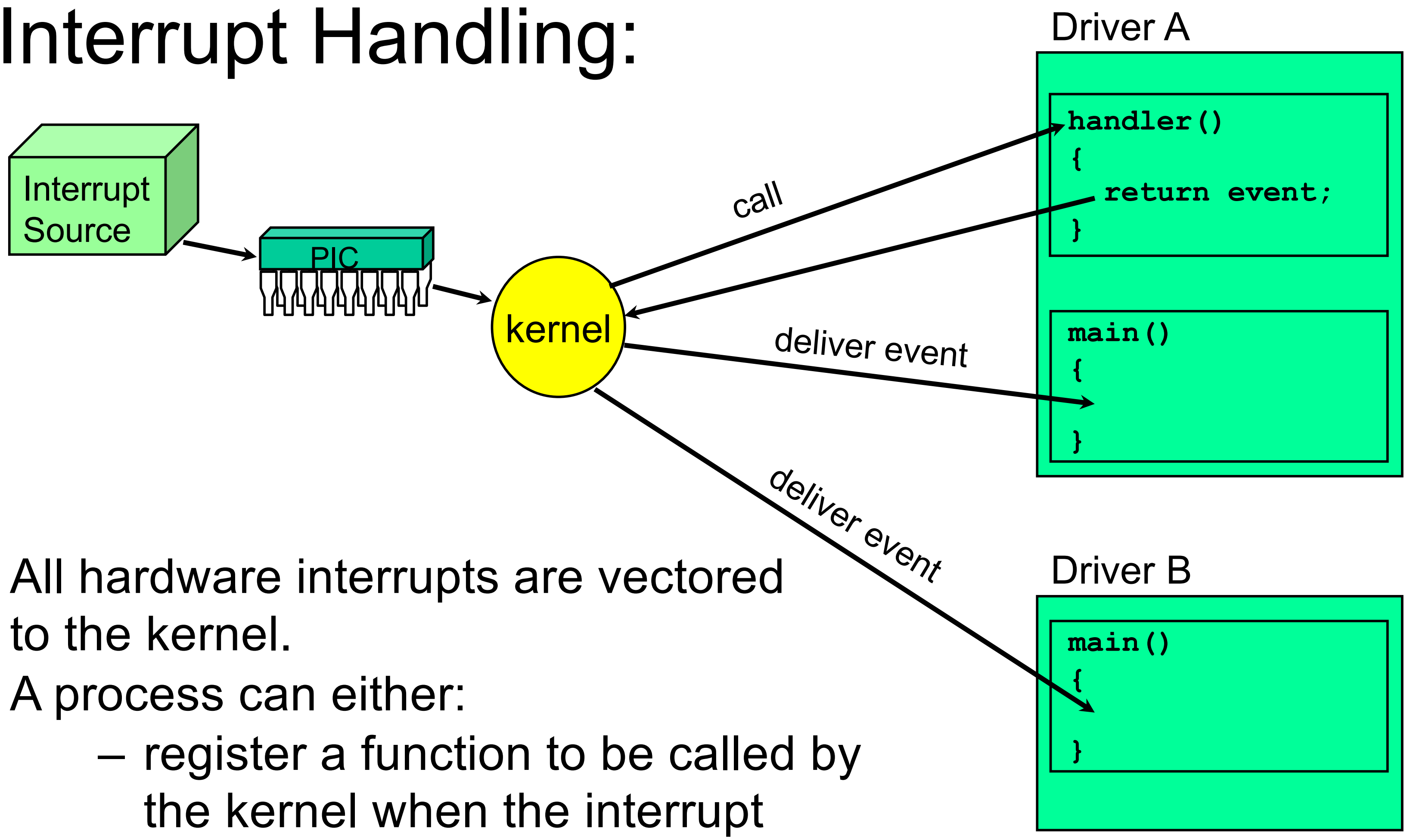
Thread synchronization methods:

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

QNX® Neutrino®'s Concept of Time:



Interrupt Handling:



All hardware interrupts are vectored to the kernel.

A process can either:

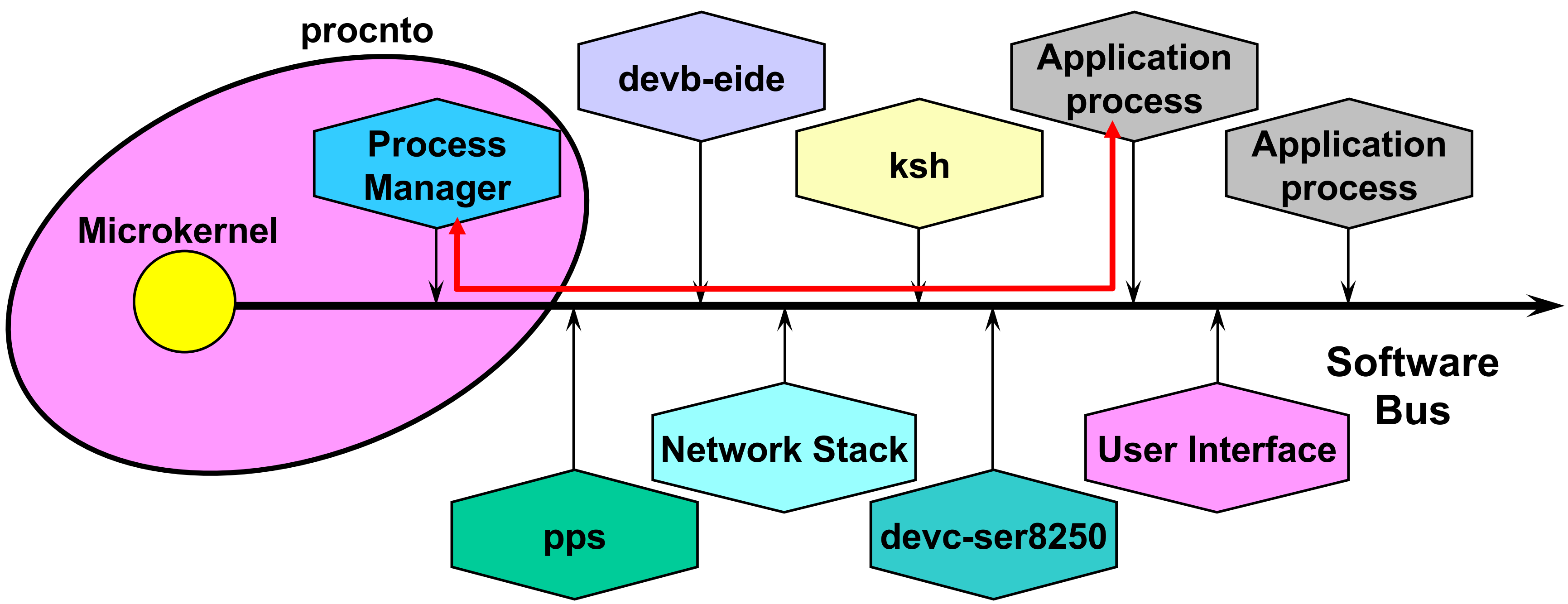
- register a function to be called by the kernel when the interrupt happens
- request notification that the interrupt has happened

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
- on a multicore system, the kernel runs where it needs to:
  - where a kernel call was made
  - where an interrupt was directed
  - where a fault happened

Topics:

Communication with the Process Manager:

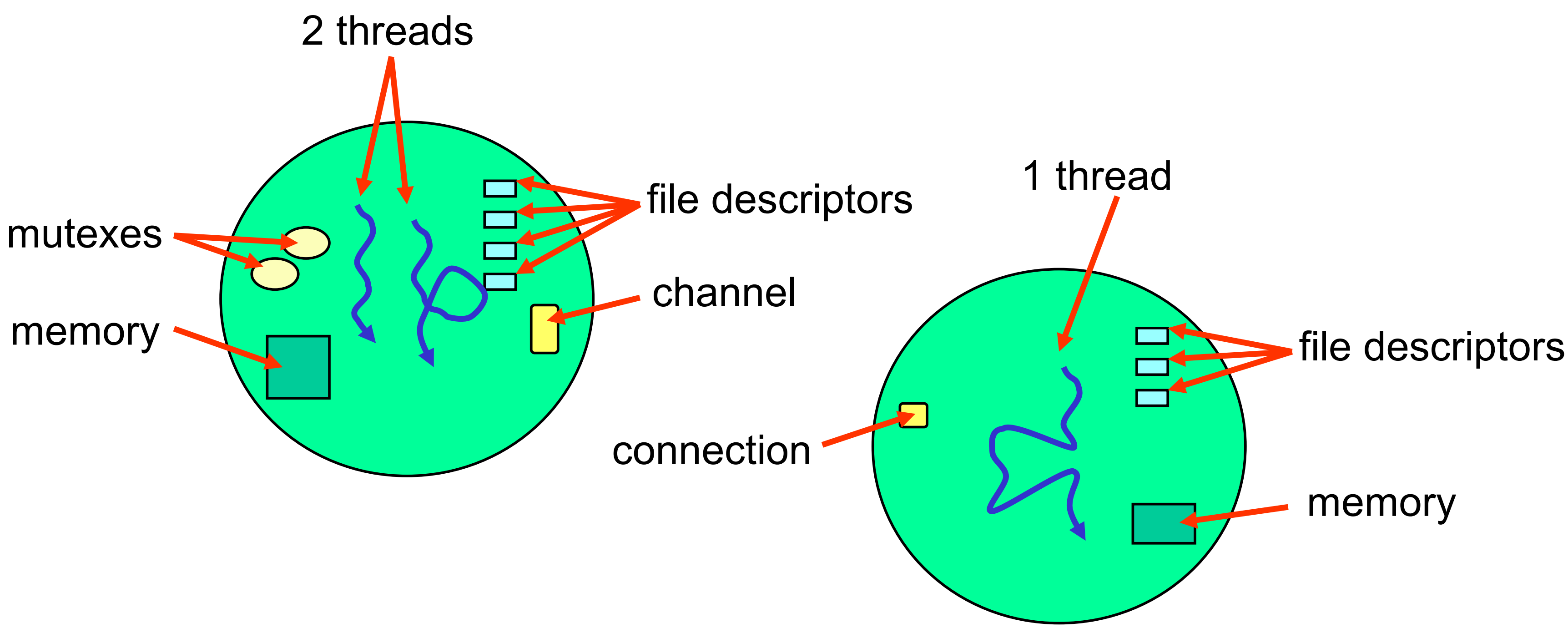


- **procnto** is QNX
  - proc for the process manager
  - nto for the Neutrino microkernel
  - they share address space, but behave differently
- process manager is reached using messages

The Process Manager provides:

- packaging of groups of threads together into processes
  - process creation and termination
    - posix_spawn / exec / fork
    - loads ELF executables
- memory protection, address space management
  - this gives shared memory for IPC
- pathname management
  - in QNX, this is process location
- several resource managers
- system state change notifications
- system information
- an idle thread per core that uses CPU no-one else wants

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

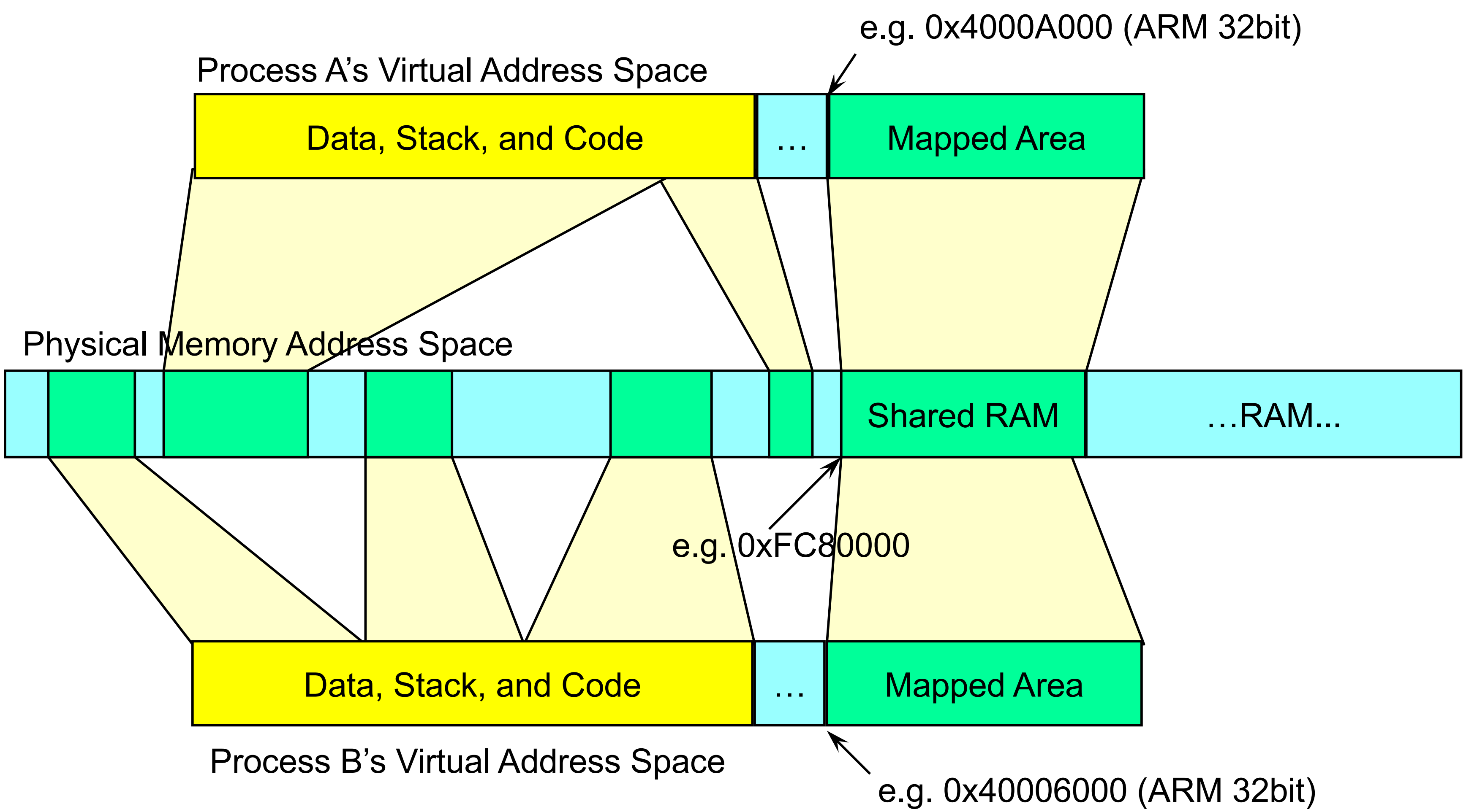


– let's look at memory in more detail…

We use a virtual address model:

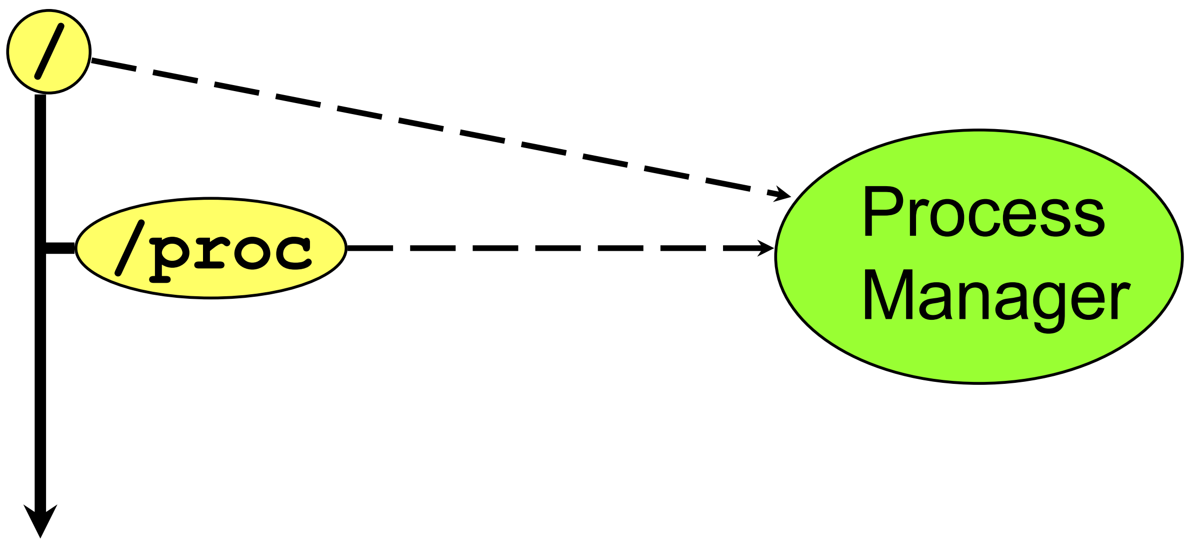
- all processes run in their own virtual address space
- all pointers/addresses you use will be virtual addresses, not physical
  - to access physical (hardware) addresses, you must create a virtual mapping
  - all processes share the underlying physical address space
- the system process/kernel (`procnto`) has an address space that doesn't overlap user processes
  - this makes kernel calls cheaper
  - on 64-bit architectures, user space is 0-512G and system space is 512G at a high address
  - on ARM 32-bit, user space is 0-2G and system is 2G-4G

Virtual addresses map to physical addresses:



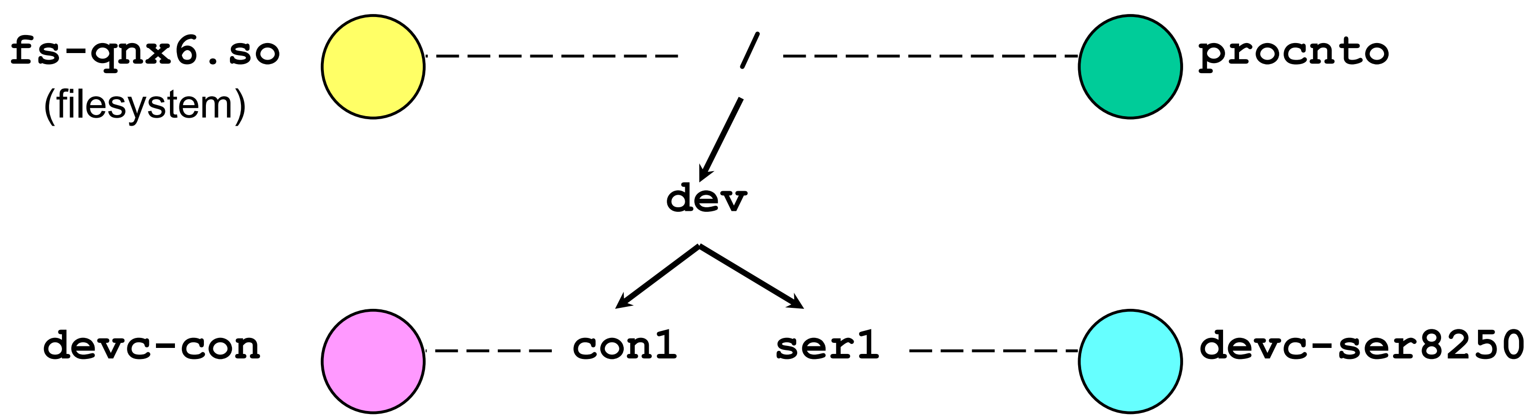
Pointers that you deal with contain virtual addresses, not physical

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`**.

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



Topics:

- Overview
- The Microkernel
- The Process Manager
- → Scheduling
- Resource Managers
- System Library
- Shared Objects
- OS Services
- Security
- Boot Sequence
- Conclusion

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

runnable

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

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 preemptive)
  - the thread's state becomes **`RUNNING`**
  - blocked threads don't even get considered
  - on a multicore system, other threads will also be running
    - we don't guarantee which, just the single highest-priority one
- most threads spend most of their time blocked
  - that is how CPU is shared between threads

Priorities:

Priorities

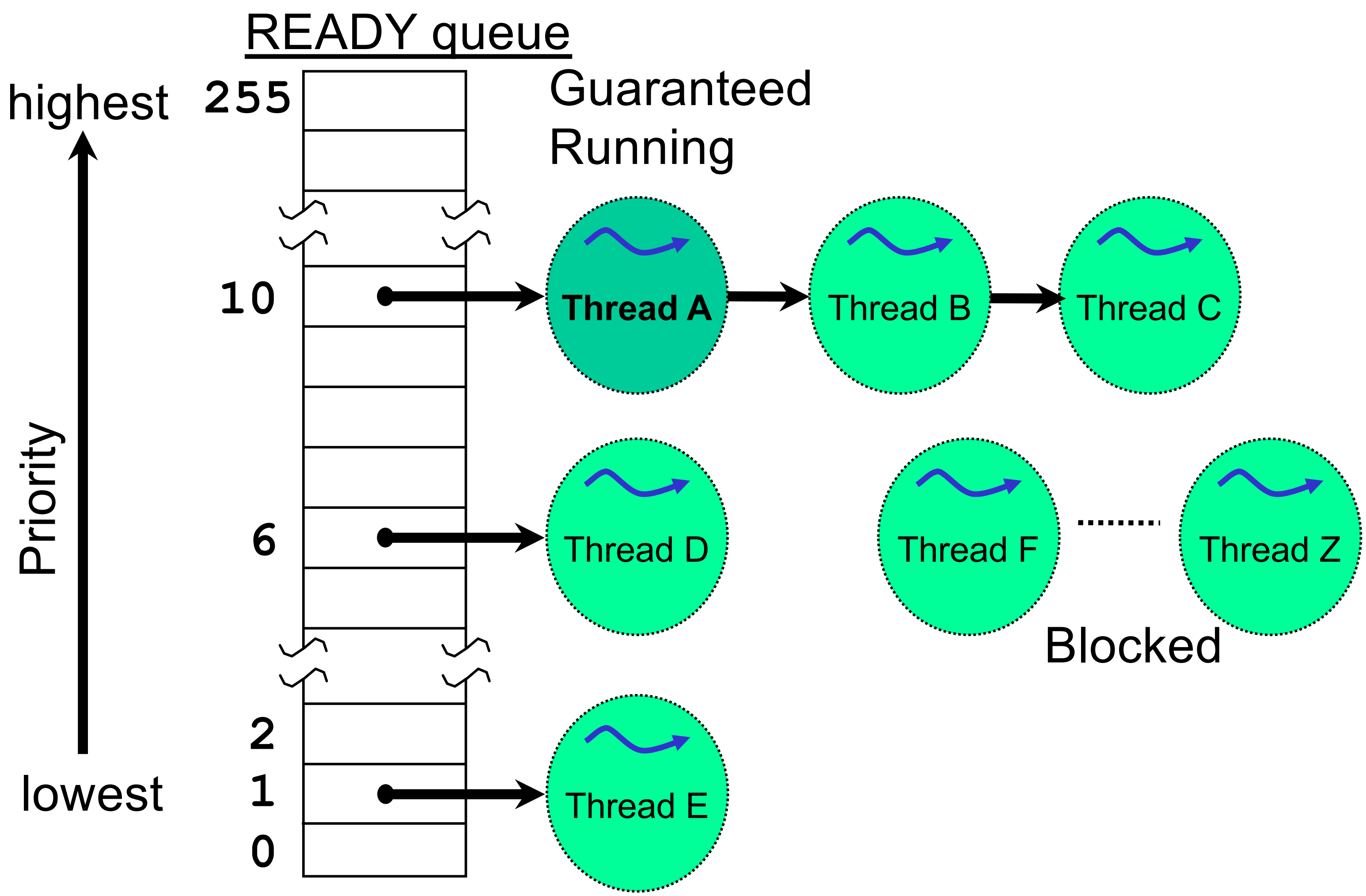
Priority

highest

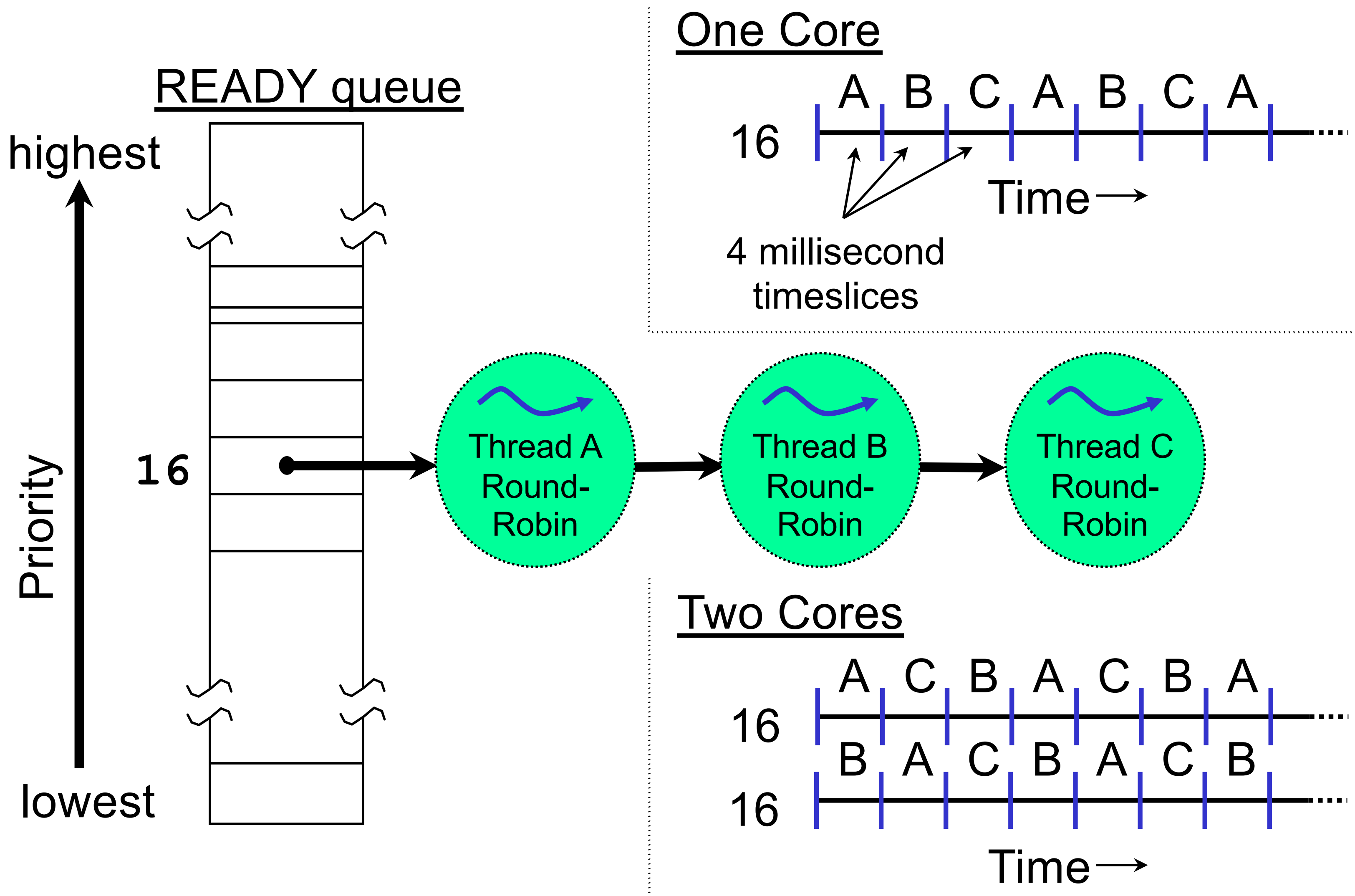
```
(hardware interrupt handlers)
-----------------------------
255
 .
 .
21  devb-eide (one of many of
                 devb-eide's threads)
 .
 .
11
10  ksh
 9
 .
 .
 1
 0  procnto (idle thread[s])
```

lowest

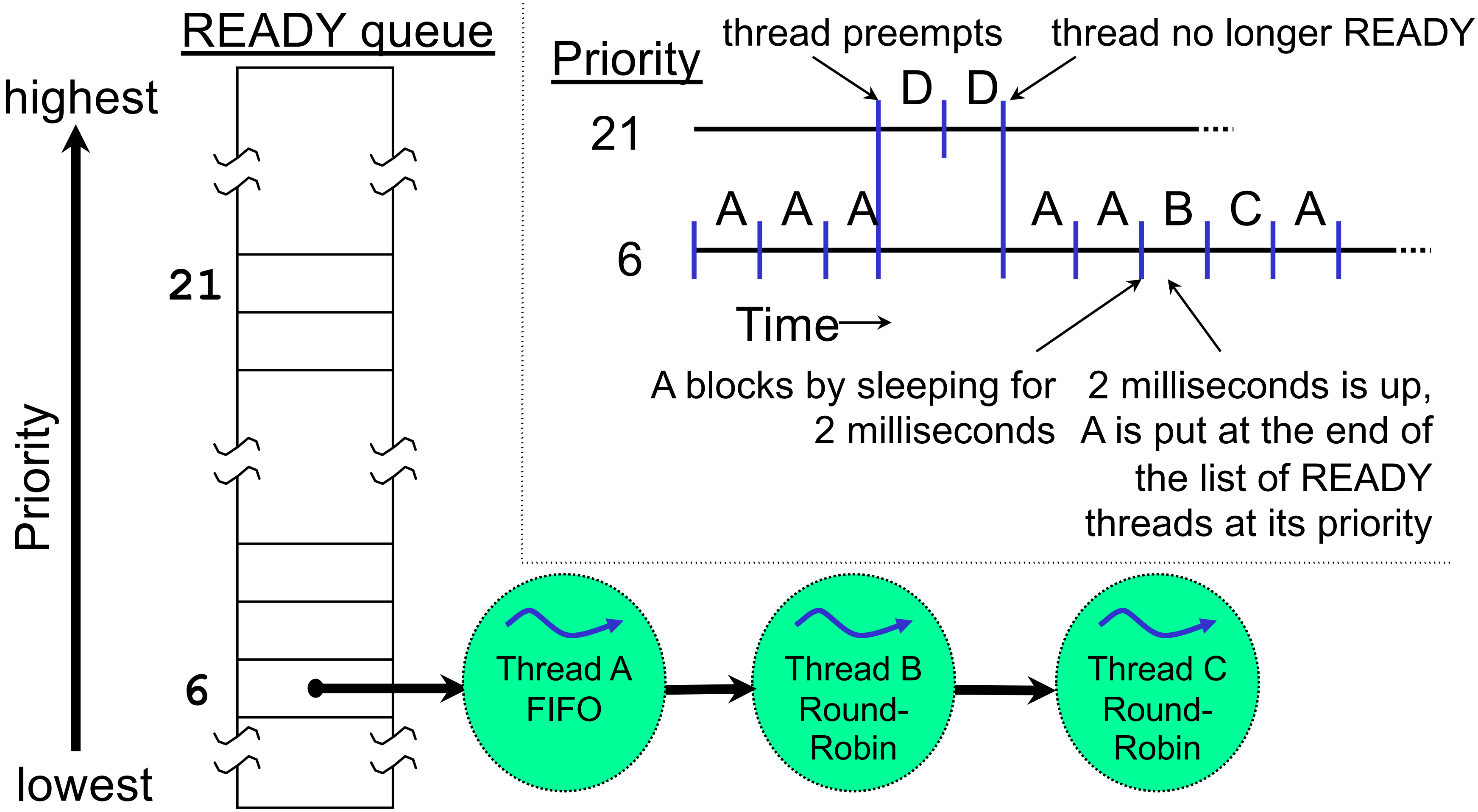
The READY queue



Scheduling algorithms: Round-robin

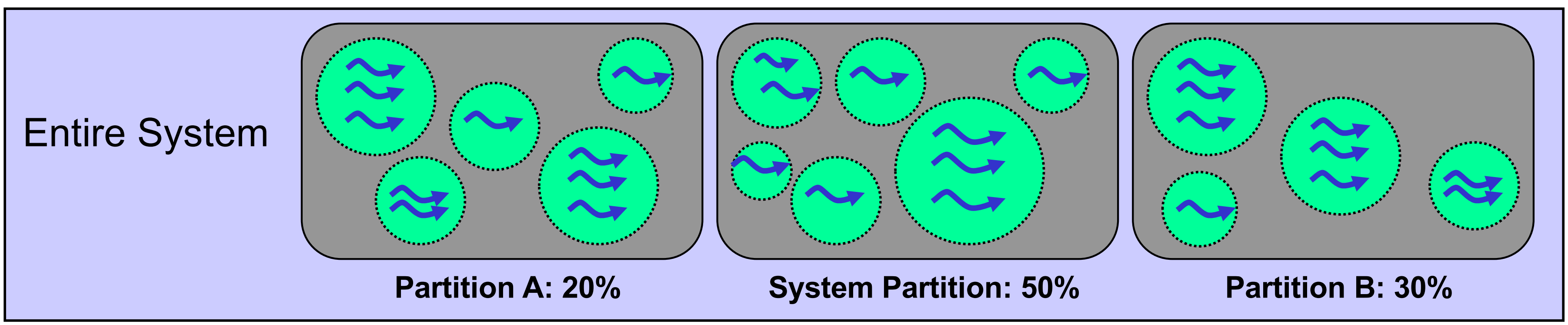


Scheduling algorithms: FIFO



System designer:

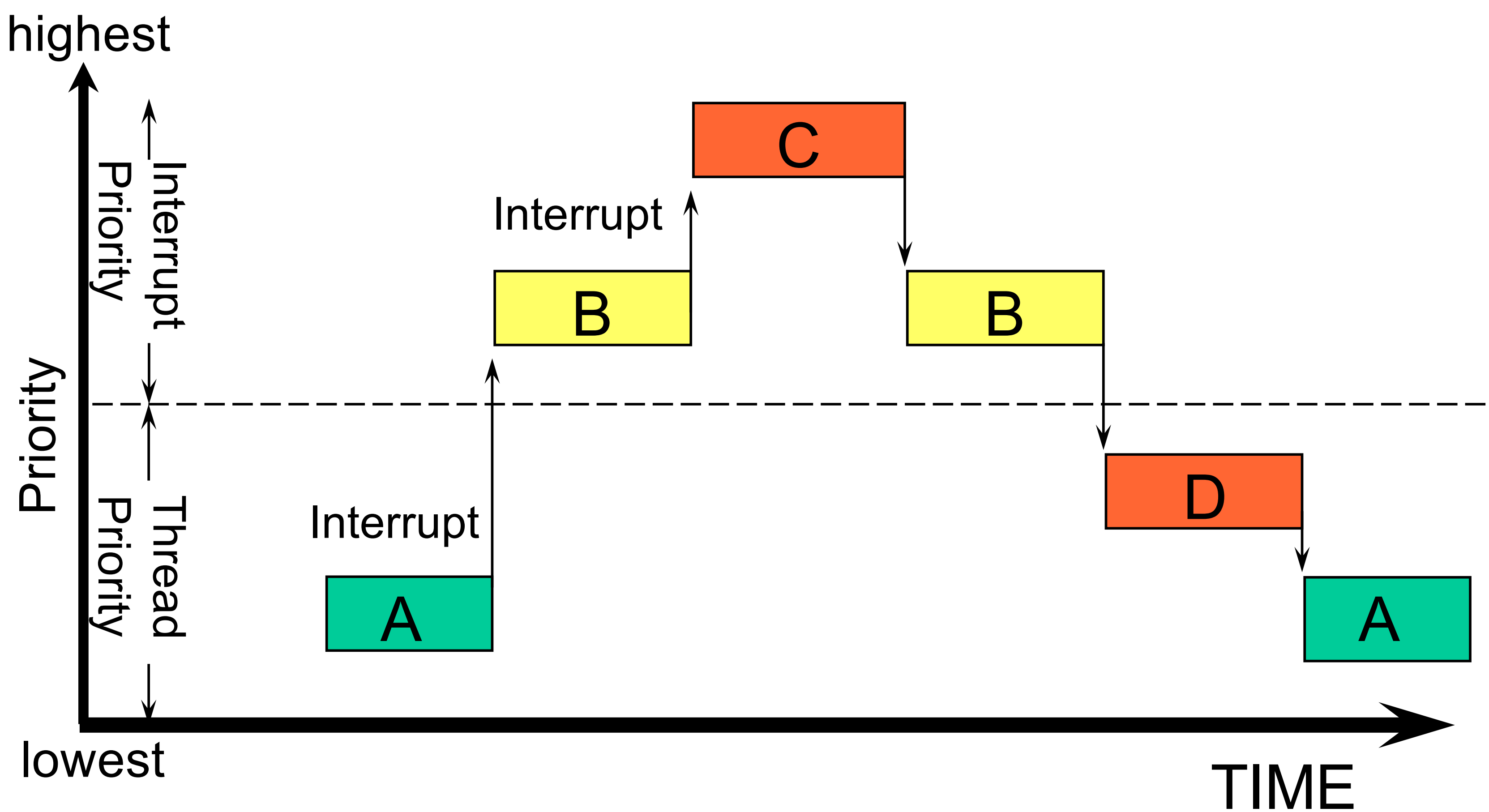
- creates scheduling partitions
- decides which partition processes/threads go into
  - child processes/threads go into parent’s partition by default
- specifies minimum % CPU usage for each partition



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 having a ‘critical priority’
  - e.g. interrupt handling threads
  - all threads that are at or above the defined ‘critical priority’ are considered to be critical
  - a critical budget (in milliseconds) must be specified (default of 0) in addition to the partition’s ‘regular’ budget
  - critical threads only deduct from their critical budget when the partition’s ‘regular’ budget has been exhausted

Interrupt Scheduling (preemptive):



Topics:

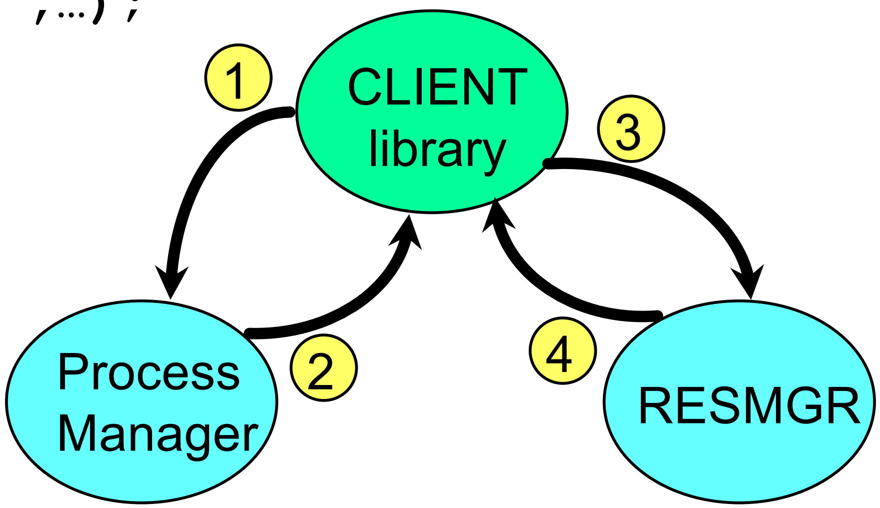
- Overview
- The Microkernel
- The Process Manager
- Scheduling
- → Resource Managers
- System Library
- Shared Objects
- OS Services
- Security
- Boot Sequence
- Conclusion

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)

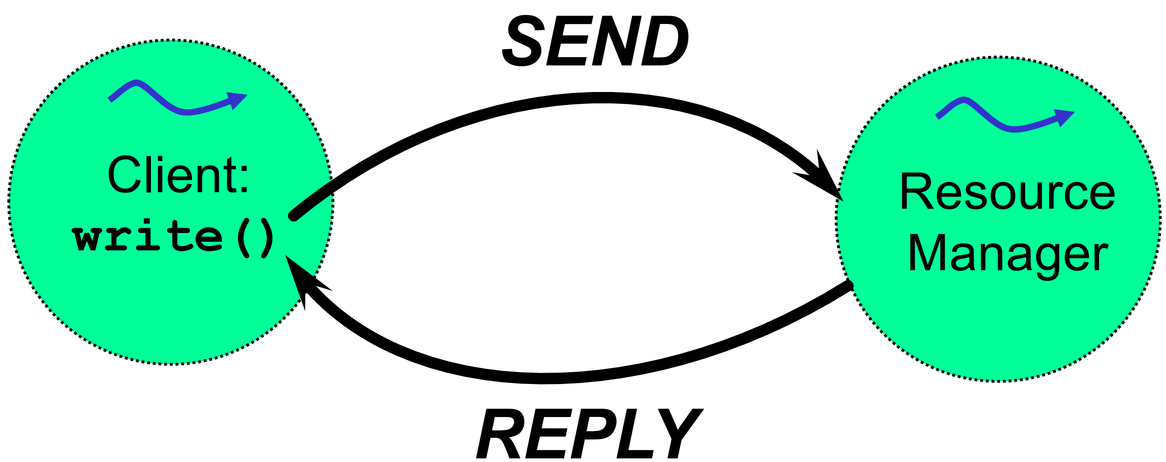
Interactions:

```
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)

Further communication is message passing directly to the resource manager:



Other notes:

- this setup allows for a lot of powerful solutions
  - provide resiliency or redundancy of OS services
  - drivers are processes, so you can:
    - debug “OS” drivers with a high-level (symbolic) debugger
    - export access to your custom driver with a network file system such as NFS or CIFS
- the QNX C library provides a lot of useful code to minimise the work needed to write one

Topics:

- Overview
- The Microkernel
- The Process Manager
- Scheduling
- Resource Managers
- → System Library
- Shared Objects
- OS Services
- Security
- Boot Sequence
- Conclusion

- 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

But QNX is a microkernel

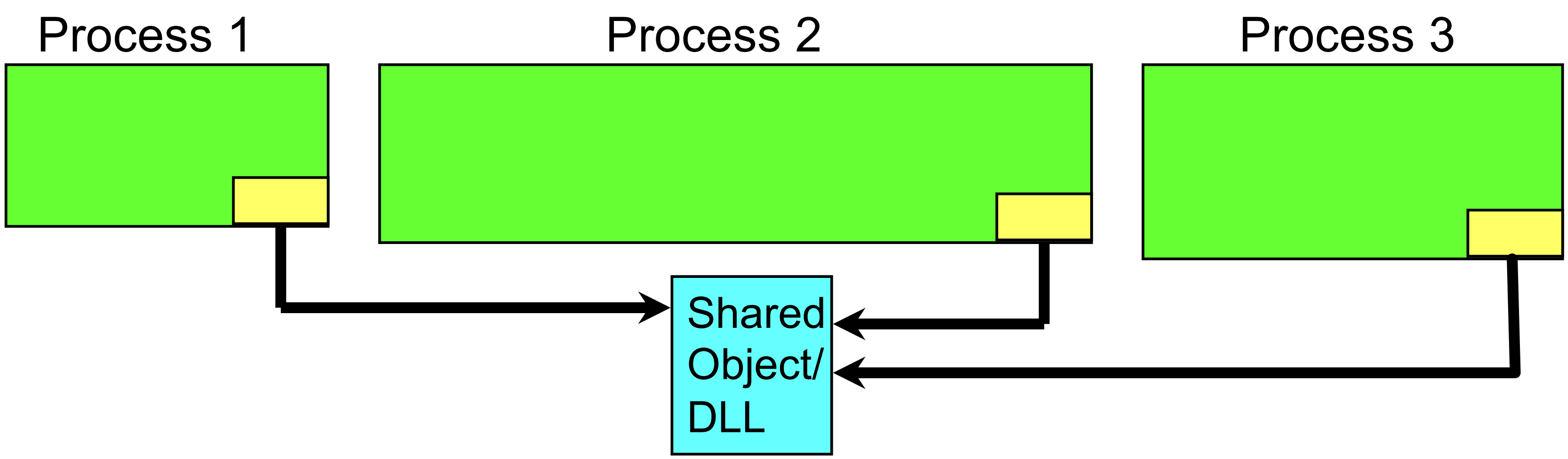
- so many routines that would be a kernel call, or have a dedicated kernel call in a traditional Unix become a message pass
- they build a message then call *MsgSend()* passing it to a server, e.g.
  - *read()* builds a message then sends it to a resource manager
  - *fork()* builds a message and sends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

Topics:

- Overview
- The Microkernel
- The Process Manager
- Scheduling
- Resource Managers
- System Library
- → Shared Objects
- OS Services
- Security
- Boot Sequence
- Conclusion

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



Topics:

- Overview
- The Microkernel
- The Process Manager
- Scheduling
- Resource Managers
- System Library
- Shared Objects
- → OS Services
- Security
- Boot Sequence
- Conclusion

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

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
`slogger2`	QNX system logger
`syslogd`	Unix syslog support
`pci-server`	PCI bus access and configuration

Topics:

- Overview
- The Microkernel
- The Process Manager
- Scheduling
- Resource Managers
- System Library
- Shared Objects
- OS Services
- → Security
- Boot Sequence
- Conclusion

QNX uses Unix style permissions:

- for files, directories, and devices
- user/group/other for read, write, and execute/search
- from the boot image (`ifs`), everything runs as root (uid 0)
  - this can be changed with:
    - launcher programs
    - `login`
    - *setuid()* and related C APIs
    - setuid executables
- `qconn` runs, and launches, everything as root by default
  - should never be running on a released system

Controlling system privileges:

- system privileges, e.g.:
  - changing user id
  - killing other user's processes,
  - accessing hardware
- traditionally controlled on a root/non-root basis
- QNX uses procmgr abilities for finer-grained control
  - ability defaults differ for root and non-root processes
  - for backwards compatibility, root (uid 0) defaults to having all abilities
  - can be controlled through the *procmgr_ability()* function
    - but, use security policies instead…

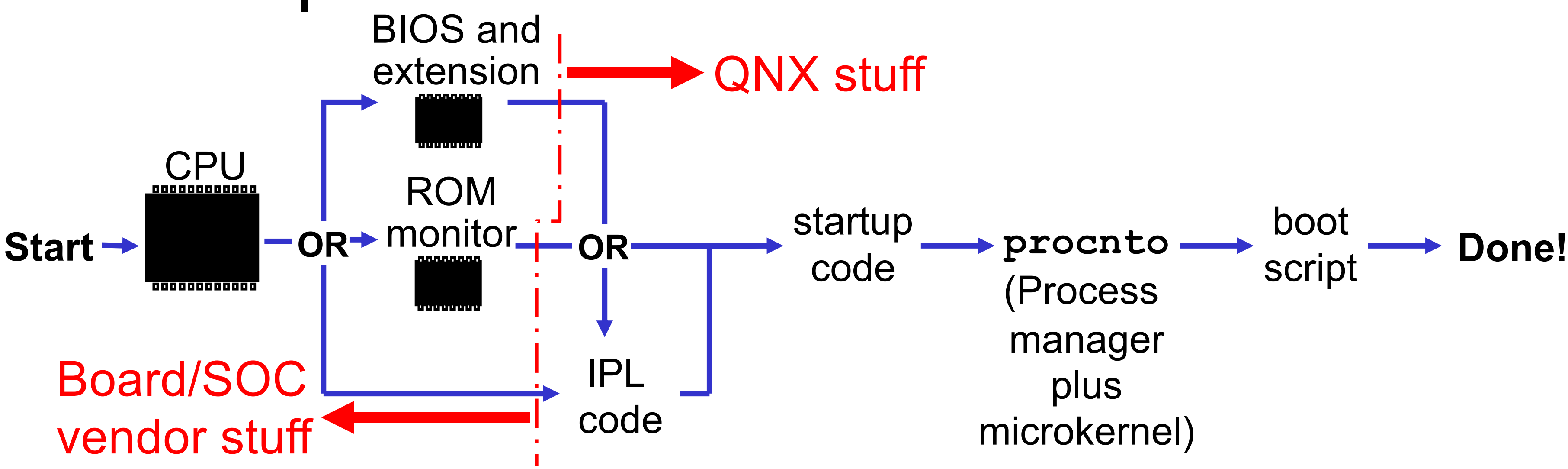
On secured systems, system privilege is controlled by security policies:

- policy is written & compiled on host
  - compiled policy loaded into kernel at boot time
- defines process types
  - what abilities a process type has
  - what transitions (to other types) are allowed
- during system initialization, all process are started with a type
- separates system privileges (type, abilities) from file/device/directory access (uid, gid)

Topics:

- Overview
- The Microkernel
- The Process Manager
- Scheduling
- Resource Managers
- System Library
- Shared Objects
- OS Services
- Security
- → Boot Sequence
- Conclusion

Boot sequence:



- IPL code:
  - does chip selects and sets up RAM, then jumps to startup code
- startup code:
  - sets up some hardware and prepares environment for `procnto`
- `procnto`:
  - sets up kernel and runs boot script
- the boot script contains:
  - drivers and other processes, including yours

The `startup` code:

- is board-specific
- tells `procnto` about core hardware, e.g.:
  - system RAM amount and layout
  - interrupt controller(s)
  - timer chip
  - special memory regions (e.g. graphics memory)
- communicates this data through the system page (syspage)
  - mapped read-only into every process

Topics:

- Overview
- The Microkernel
- The Process Manager
- Scheduling
- Resource Managers
- System Library
- Shared Objects
- OS Services
- Security
- Boot Sequence
- → Conclusion

Conclusion

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
- drivers are processes
- QNX Neutrino does preemptive scheduling
  - only `READY` threads are schedulable, blocked threads are not