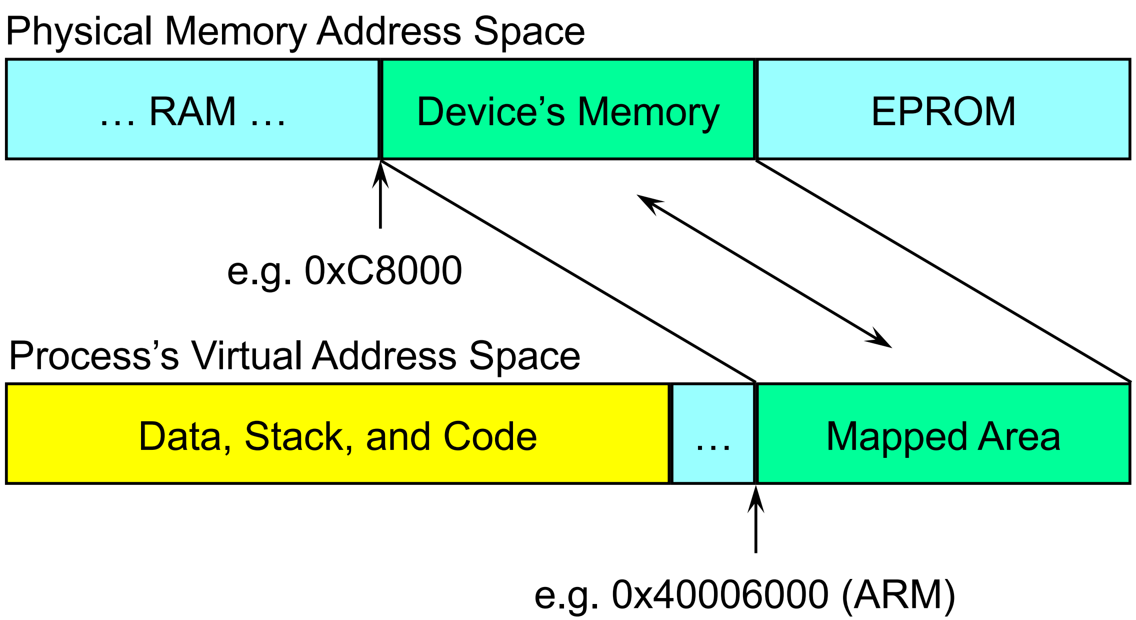
Introduction to Hardware Programming

Topics:

- → **Hardware I/O**
- **Programming PCI bus devices**
- **Handling Interrupts**
- **Conclusion**

To access memory on a hardware device:

– physical addresses must be mapped into your process's virtual address space:




```
// get a pointer to memory physically located at 0xc8000
vaddr = mmap_device_memory (0, 0x4000,
                    PROT_READ | PROT_WRITE | PROT_NOCACHE,
                    0, 0xc8000);
```

DMA operations usually require physically contiguous RAM:

```
// for DMA:
// allocate len bytes of physically contiguous
// system memory

vaddr = mmap (0, len,
              PROT_READ | PROT_WRITE | PROT_NOCACHE,
              MAP_PHYS | MAP_ANON | MAP_SHARED, NOFD, 0);

// get the physical address for passing to the controller
mem_offset64 (vaddr, NOFD, len, &paddr, NULL);
```

Control Registers

How you access control registers varies based on the platform:

- on ARM and AARCH64 platforms:
  - registers are mapped like memory
  - pointer dereferences are used for read/write operations
  - procnto knows based on system page that these are not RAM, sets CPU access flags differently
  - some x86_64 devices may, also, be accessed this way
- on x86_64 most devices continue to use “x86-style” I/O ports
  - a special address line must be active to change from memory to I/O addressing
  - requires a special set of assembly instructions for access

Interfacing to I/O ports (x86_64):

- QNX supplies cover functions for the inline assembly needed

```
#include <hw/inout.h> // header for in*() & out*() fns

// enable I/O privilege for this thread
ThreadCtl (_NTO_TCTL_IO, NULL);

val8  = in8  (ioport_addr); // read an 8 bit value
val16 = in16 (ioport_addr); // read a 16 bit value
val32 = in32 (ioport_addr); // read a 32 bit value

out8  (ioport_addr, val8 ); // write an 8 bit value
out16 (ioport_addr, val16); // write a 16 bit value
out32 (ioport_addr, val32); // write a 32 bit value
```

Topics:

- Hardware I/O
- → Programming PCI bus devices
- Handling Interrupts
- Conclusion

To find and configure a PCI device:

- you must run the PCI server:

```
pci-server
```

The PCI calls include:

`pci_device_find()`	find hardware by Device ID and Vendor ID
`pci_device_attach()`	find hardware and get basic configuration information
`pci_device_reset()`	reset device
`pci_device_detach()`	detach device
`pci_device_read_*()`	read configuration information
`pci_device_write_*()`	write to device command or status register
`pci_device_map_as()`	translate between CPU and PCI addresses

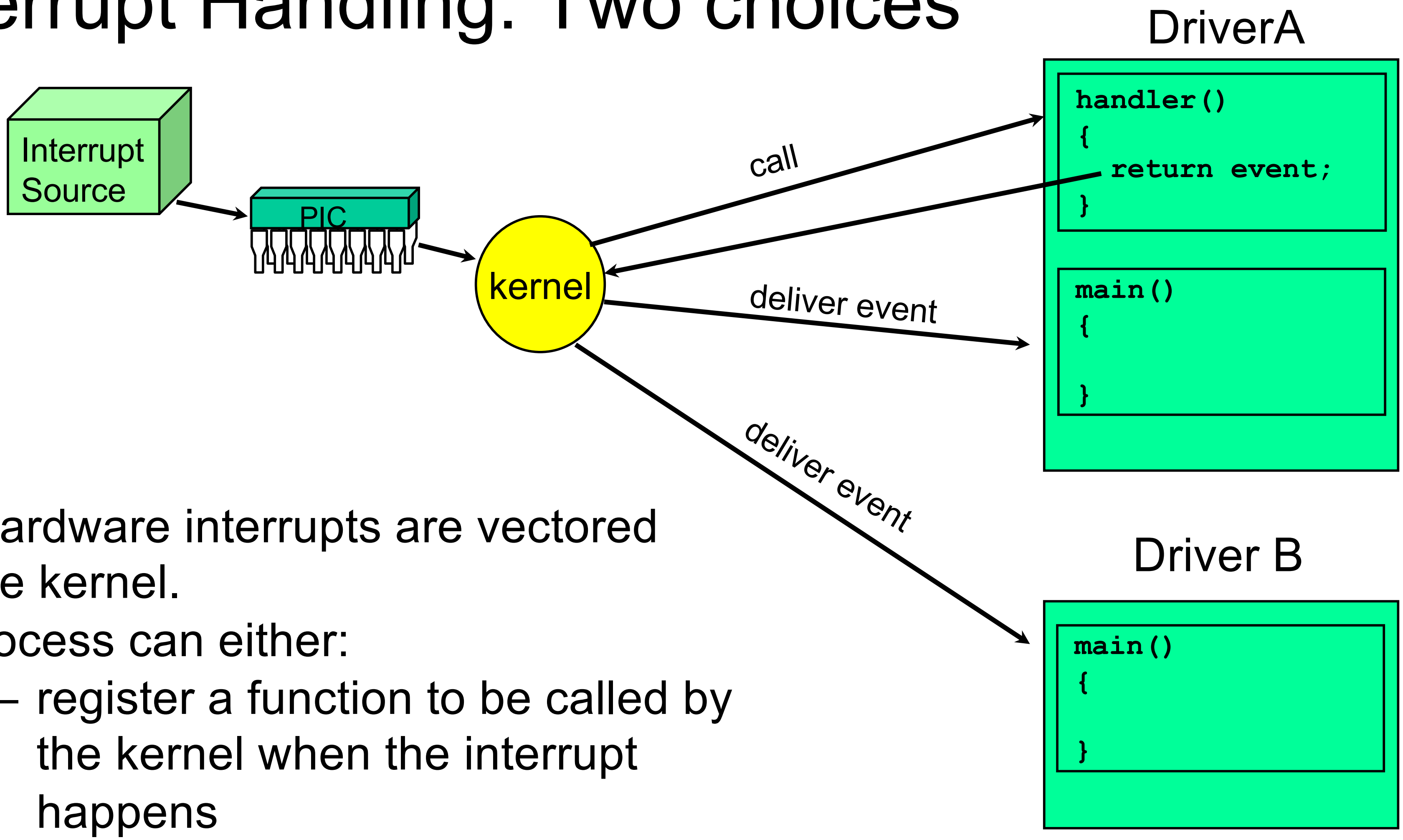
The *pci_device_find()* call:

- fills in a **`pci_bdf_t`** base data type
  - encodes the bus, device and function of the PCI device
- various calls can determine
  - interrupt number
  - address translations for bus master PCI devices
  - PCI configuration space registers
  - the base addresses of a device

Topics:

- Hardware I/O
- Programming PCI bus devices
- → Handling Interrupts
- Conclusion

Interrupt Handling: Two choices



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

Interrupt calls:

```
id = InterruptAttach (int intr,
                      struct sigevent *(*handler)(void *, int),
                      void *area, int size, unsigned flags);
id = InterruptAttachEvent (int intr, struct sigevent *event,
                           unsigned flags);
InterruptDetach (int id);
InterruptWait (int flags, uint64_t *reserved);
InterruptMask (int intr, int id);
InterruptUnmask (int intr, int id);
InterruptLock (struct intrspin *spinlock);
InterruptUnlock (struct intrspin *spinlock);
```

☞ Permissions are complicated… see next slide.

Interrupt Calls – System Privileges

The privileges required for interrupt calls are complex:

- *InterruptAttach()*
  - **`PROCMGR_AID_INTERRUPT`**
- *InterruptAttachEvent()*
  - **`PROCMGR_AID_INTERRUPTEVENT`** or **`PROCMGR_AID_INTERRUPT`**
- *InterruptMask()*, *InterruptUnmask()*
  - I/O privilege is required to mask an interrupt the process has not attached
- *InterruptLock()*, *InterruptUnlock()*, *InterruptDisable()*, *InterruptEnable()*
  - I/O privilege which is gotten by calling:
- *ThreadCtl(_NTO_TCTL_IO, 0)*:
  - **`PROCMGR_AID_IO`**

Driver A example: interrupt handler function

```
struct sigevent event;

const struct sigevent *
handler (void *not_used, int id)
{
  if (check_status_register())
    return (&event);
  else
    return (NULL);
}
main ()
{
  SIGEV_INTR_INIT (&event);
  id = InterruptAttach (INTNUM, handler, NULL, 0, ...);
  for (;;) {
    InterruptWait (0, NULL);
    // do some or all of the work here
  }
}
```

Driver B example: interrupt event loop

```
struct sigevent event;

main ()
{
  SIGEV_INTR_INIT (&event);
  id = InterruptAttachEvent (INTNUM, &event, ...);
  for (;;) {
    InterruptWait (0, NULL);
    // do the interrupt work here, at thread priority
    InterruptUnmask (intnum, id);
  }
}
```

Telling kernel what code to run when an interrupt happens:

```
id = InterruptAttachEvent (intr, event, flags);
```

tell kernel what wakeup
event to give us

logical interrupt
vector number

additional
information
flags

handler function
to run on interrupt

mem passed
Into handler

```
id = InterruptAttach (intr, handler, area, size, flags);
```

InterruptAttach*() Flags

InterruptAttachEvent() and *InterruptAttach()*'s flags parameter can contain:

- **_NTO_INTR_FLAGS_END:** If multiple interrupt handlers, specify we should execute last
- **_NTO_INTR_FLAGS_PROCESS:** for events types that are directed at a process rather than a thread, e.g. pulses
- **_NTO_INTR_FLAGS_TRK_MSK:** request that the kernel adjust the interrupt mask when the attaching process terminates (ALWAYS set this flag)
- **_NTO_INTR_FLAGS_NO_UNMASK**: must explicitly unmask the interrupt to enable

An interrupt handler operates in the following environment:

- it is sharing the data area of the process that attached it
- the environment is very restricted:
  - cannot call kernel functions except *InterruptMask()*, *InterruptUnmask*() and *TraceEvent()* (see notes)
  - cannot call any function that might call a kernel function
    - the documentation for each function specifies whether or not that function is safe to call from an interrupt handler
    - there is also a section in the Library Reference manual called “Full Safety Information” that lists all safe functions
  - the interrupt handler is using the kernel’s stack, so keep stack usage small (if you have a lot of data, use variables defined outside of the handler rather than variables defined local to the function, and don’t call too many function levels deep)
  - shouldn’t do floating point or equivalent operations

Should you attach a handler or an event?

- The kernel is the single point of failure for a QNX system; attaching a handler increases the size of the SPOF, an event does not
- debugging is far simpler with an event
  - ISR code can not be stepped/traced with the debugger
- full OS functionality when doing h/w handling in a thread
- events impose far less system overhead at interrupt time than handlers
  - no need for the MMU work to gain access to process address space if using an event
- scheduling a thread for every interrupt could be more overhead, if you could do some work at interrupt time and only need to schedule a thread some of the time
- handlers have lower latency than getting a thread scheduled
  - does your hardware have some sort of buffer or FIFO? If not, then you might not be able to wait until a thread is scheduled

You have the following notification methods:

- SIGEV_INTR
  - unblocks *InterruptWait()*
  - simplest to use (least setup)
  - fastest (lowest overhead, latency)
  - not queued or counted
  - must dedicate a thread
- SIGEV_SEM
  - unblocks *sem_wait()* on a named semaphore
    - within driver, use an anonymous named semaphore
  - counted
  - only direct cross-process choice

continued...

Notification methods (continued):

- SIGEV_SIGNAL
  - unblocks *sigwaitinfo()*
    - Do not use a signal handler, the overhead and latency are awful
  - can be queued
  - can carry data
- SIGEV_PULSE
  - unblocks *MsgReceive*()*
  - queued
  - carries data
  - most flexible
    - single threaded driver can handle hardware and clients
    - pool of threads
  - highest overhead and latency

Simple interrupt handler:

- in your **`interrupt`** project is a skeleton file called **`intsimple.c`**
- fill it in with the code for handling interrupts, the instructor will tell you which interrupt to attach to
- attach an interrupt handler that will return a **`SIGEV_INTR`** event
- in the loop, use *InterruptWait()* to wait for the interrupt notification

Topics:

Hardware I/O

Programming PCI bus devices

Handling Interrupts

→ Conclusion

Conclusion

You learned:

- That memory or port mappings have to be set up to access hardware devices
- that the kernel is the first handler for all interrupts
- that processes can register handlers or can register for notification of interrupts
- that interrupt handlers run in a very restricted environment