What’s in a device driver?

Role of the OS

- Protect data and resources (file permissions, etc.)
- Provide isolation (virtual memory, etc.)
- Abstract away details of hardware
  - Processes use “stdin”, not keyboard I/O ports
  - Processes use open/read/write/close, not disk seeks
- Multiplex hardware resources
  - Processes think they own CPU, kbd, network, etc.

Structure of the OS

- Lots of (mostly) hardware and device-independent code
  - File systems, scheduler algorithms, sockets and networking code, etc.
- Small bits of device-dependent code
  - *driver* for every specific model of…
    - network card
    - audio card
    - graphics card
    - usb device
    - …
  - each set of similar drivers provides uniform API to rest of OS

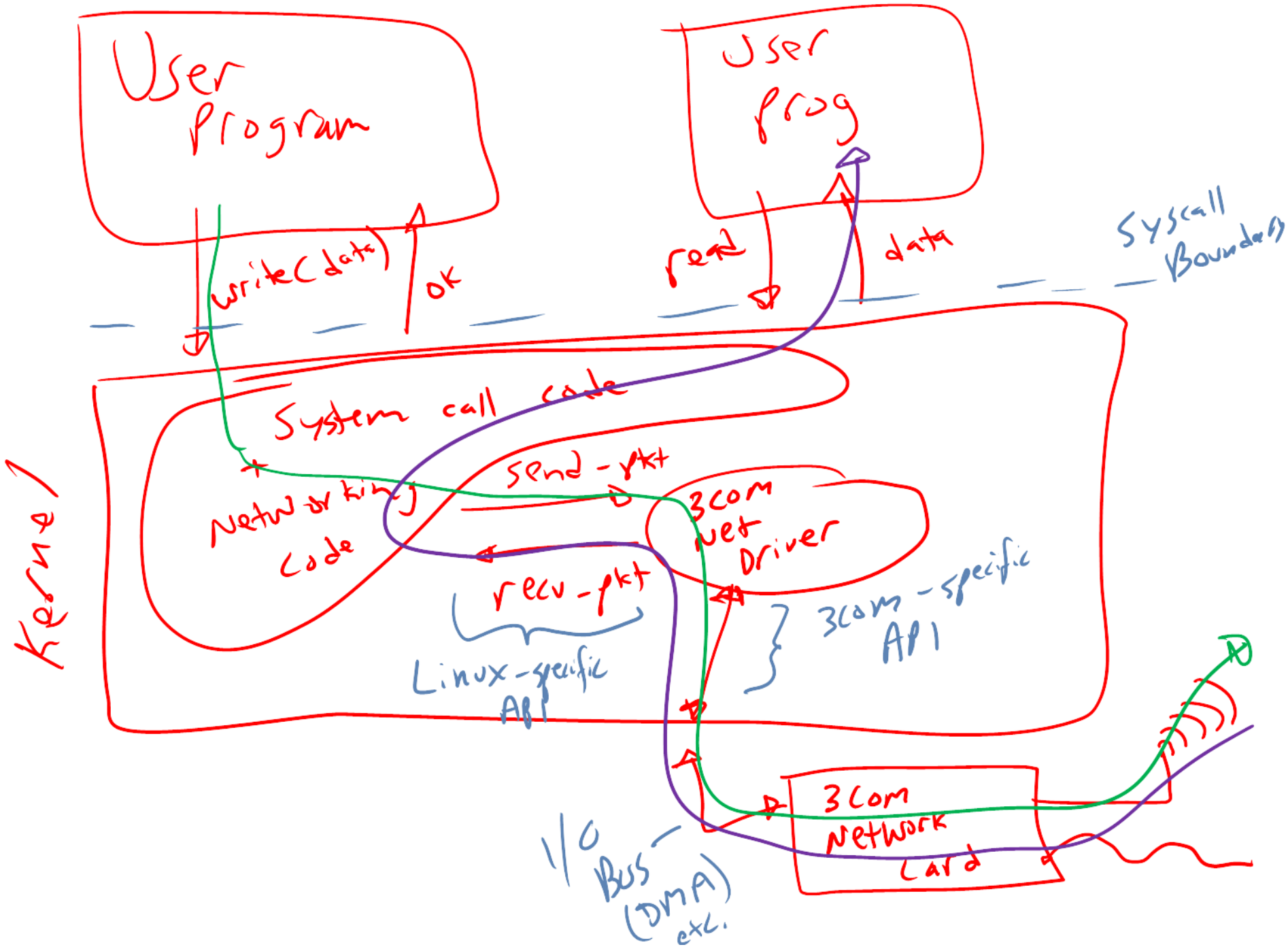
Example: Unix Network Drivers

All network drivers implement

- probe(…) – check if this device exists
- configure(…) – turn device on/off, etc.
- send_pkt(pkt) – send packet out to network

All network drivers invoke

- recv_pkt(pkt) – when packet arrives
- alloc_dma() – when they need dma’able pages


User Program
User prog
write(data)
ok
read
data
Syscall Boundary
System call code
Networking code
send-pkt
recv-pkt
3com net Driver
Linux-specific API
3com-specific API
Kernel
I/O Bus (DMA) etc.
3Com Network Card

Driver-to-Device Interaction

- Driver needs to
  - configure the device, turn it on/off, etc.
  - tell device when to send a packet
    - tell it when and where to go in memory to get packet data (via DMA from mem to device)
- Device needs to
  - tell device when a packet has arrived (e.g. via interrupts)
- Driver also needs to
  - tell device where to put incoming packets in memory (via DMA from device to mem)

Example: Broadcom tg3 network card and a driver for it from Linux

- A few years old, but very typical of many cards
- Uses DMA for ingoing and outgoing packets
- Uses interrupts to notify of packet arrival
- Uses Memory mapped I/O for configuring device, checking status, etc.

Sending a packet

- Driver implementation for send_pkt(pkt_ptr)

packet is already on
    one dma'able page

dev->dma_base = v_to_p(pkt_ptr)

dev->dma_len = pktlen

dev->cmd = SEND|DMA_ENABLE

But…

Previous would be slow

- driver needs to talk to card for every send (3 times: base, len, cmd)
- probably would have to wait for current DMA to finish before setting up next one (can't overwrite dma_base while it is being used!)

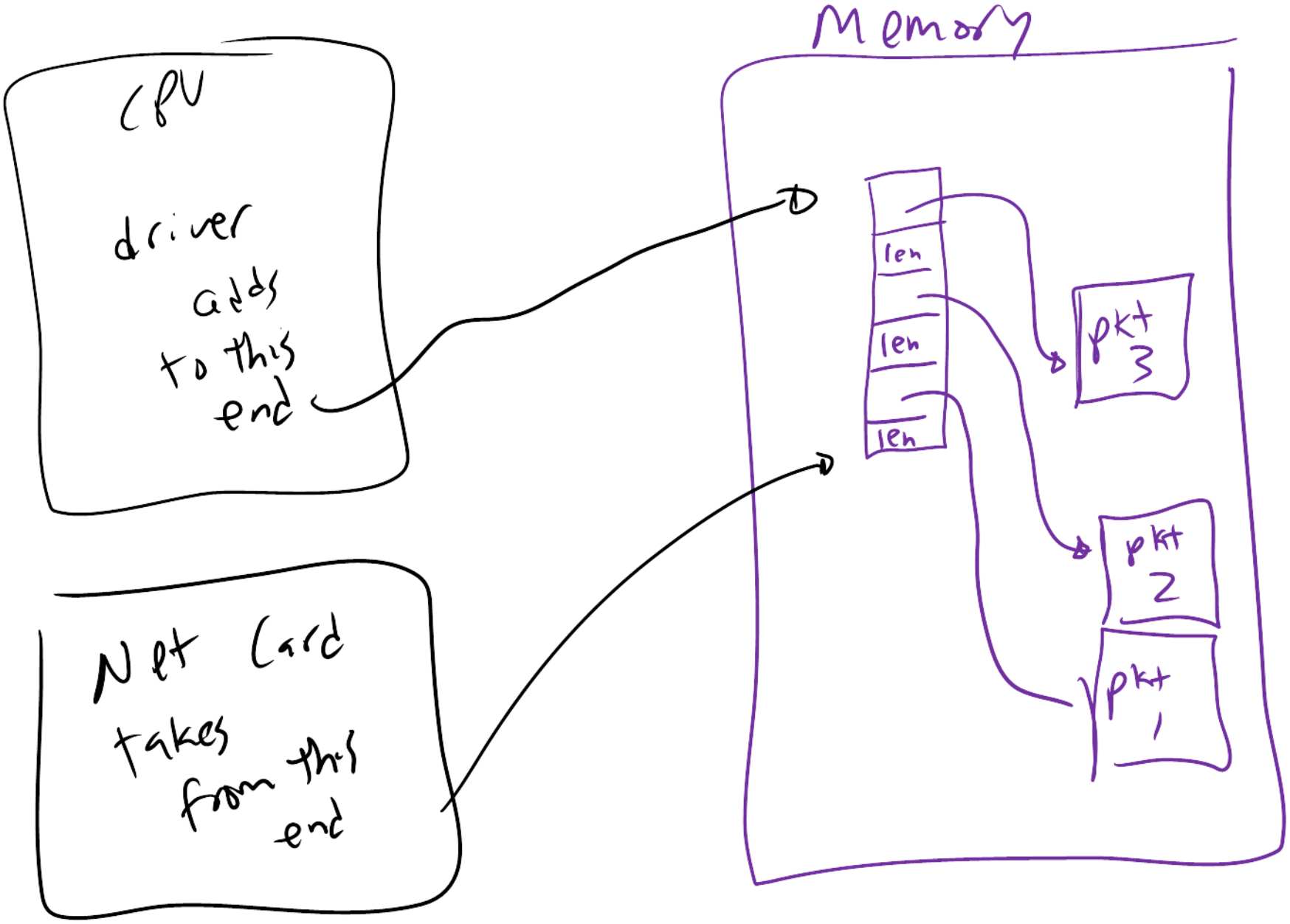
What really happens

Device and driver share a *list* of packets to send

- list is stored in memory
- driver accesses using regular inst. / code
- device accesses using DMA

Driver adds dma'able pages to the list

Device removes them after they are sent


CPU
driver
adds
to this
end
Net Card
takes
from this
end
Memory
len
len
len
pkt
3
pkt
2
pkt
1

Actually… a fixed size circular list

```
struct ring_elt { int phys_addr; int len }
struct ring {
  // all indexes are modulo 128
  int head; // index of most recently added pkt
  int tail; // index of of most recently sent pkt
  ring_elt entries[128];
}
```

driver only modifies free entries and head

device only modifies tail

Sending a packet

```
configure() { …
   ring = alloc_dma(sizeof(struct ring));
   dev->ring = v_to_p(ring)
}
send_pkt() {
  while (ring is full)
     wait();
  add v_to_p(pkt) to ring;
  increment head;
}
```

Receiving a packet?

- Same idea, but in reverse

```
configure() { …
   rx_ring = alloc_dma(sizeof(struct ring));
  tx_ring = alloc_dma(sizeof(struct ring));
  dev->rx_ring = v_to_p(rx_ring)
 dev->tx_ring = v_to_p(tx_ring)
}
net_interrupt_handler() {
  while (rx_ring is not empty) {
    get pkt off of ring
    increment tail
      recv_pkt(pkt)
  }
}
```

What does OS do?

- figure out what to do with arriving packet, then invoke corresponding function
  - (e.g. a firewall, a router, an application, …)
- or buffer received packets in a list until some other code claims them

Loose Ends: Flow control

- What if sending packets too quickly?
  - driver send_pkt() will wait until it can send it
- What if packets arriving too quickly?
  - device will DROP packets if ring is full
  - driver needs to get packets off the ring quick!
  - no time for much computation (esp. with bursty traffic) – just add received packets to a list and deal with them later

Loose Ends: Concurrency

- What if packets arrive *while* we are already processing a packet arrival interrupt?
  - Turn off interrupts for entirety of packet_arrival_interrupt_handler()

Loose Ends: Concurrency 2

- What if packets arrive while kernel is doing other stuff?
  - if interrupt handler and “other stuff” don’t touch any of the same variables, data, etc.
  - …then no problem

Loose Ends: Concurrency 3

- What if packets arrive while kernel is doing other stuff?
  - if interrupt handler and “other stuff” do touch some of the same variables, data, etc.

    e.g. kernel is looking at the recent arrivals list
  - … then we need to be careful about concurrency
- e.g. other code can disable interrupts while it looks at the recent arrivals list
- Can’t mutex / spinlock: why?
  - A: because if it is locked by other code when interrupt happens, driver will spin **forever** waiting to get into the critical section