Network Driver

OS Study Session #6

Labs
Part One: NIC

Problem Summary

- Filling the function e1000_recv() and e1000_transmit()
- Understanding overall process of sending / receiving data with NIC

Role of Network Driver (E1000)

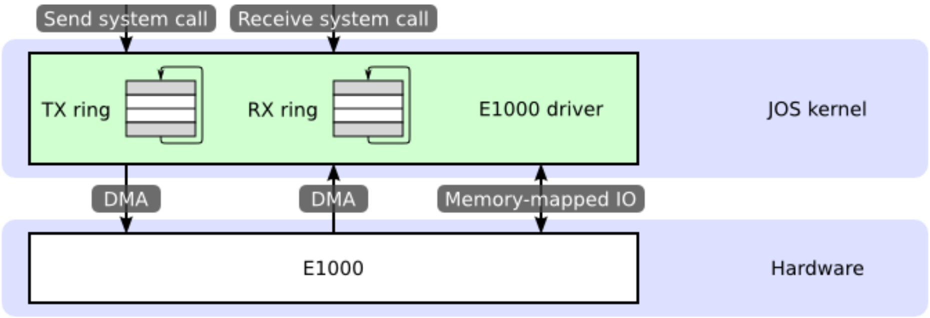
- Allocating memory for the transmit and receive queues
- Setting up DMA descriptors
- Configuring the E1000 with the location of these queues
- everything after that is asynchronous

DMA (Direct Memory Access) : read and write packet data directly from memory without involving the CPU

https://pdos.csail.mit.edu/6.828/2018/labs/lab6/

Receive System Call

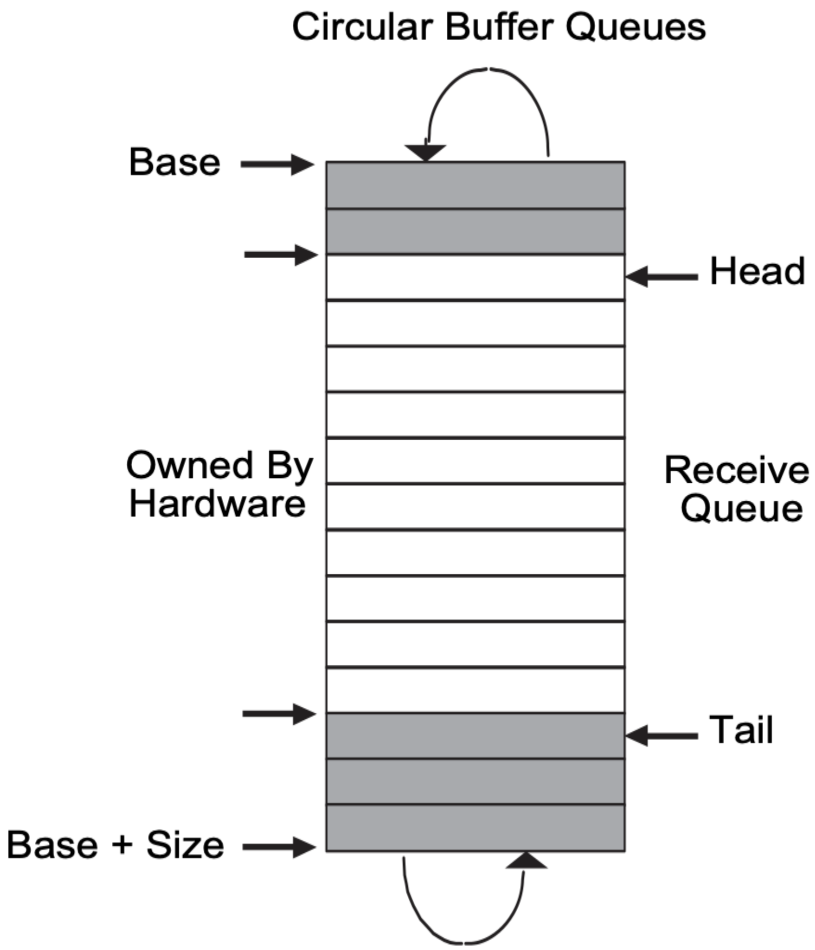
When the E1000 receives a packet, it copies it into the next DMA descriptor in the receive queue, which the driver can read from at its next opportunity.



https://pdos.csail.mit.edu/6.828/2018/labs/lab6/

1. NIC receives the new data packet in receiving buffer
2. An interrupt is triggered by NIC to notify
3. Network driver calls e1000_recv() to handling this interrupt (e1000_recv() = interrupt handler)
4. e1000_recv() transfers the data to OS's network stack

Receive Descriptor Ring Structure



- The driver always add descriptors to the tail and moves the tail pointer
- Hardware always consumes descriptors from the head and moves the head pointer
- The contents of the queue are the descriptors between head and tail pointer
- Receive queue contents : Free descriptors that the card can receive packets into
- Transmit queue content : packets waiting to be sent

https://pdos.csail.mit.edu/6.1810/2024/readings/8254x_GBe_SDM.pdf

https://pdos.csail.mit.edu/6.828/2018/labs/lab6/

e1000_recv

```
#define E1000_RXD_STAT_DD 0x01 /* Descriptor Done */
#define E1000_RXD_STAT_EOP 0x02 /* End of Packet */
```

net_rx() : deliver a packet to the networking stack

1. Find an the next waiting received packet index
2. Check if a new packet is available
3. Deliver a packet
4. Allocate a new buffer (kalloc()) to replace the one just given to net_rx().
5. Update E1000_RDT register to the new index

```
static void
e1000_recv(void)
{
  //
  // Your code here.
  //
  // Check for packets that have arrived from the e1000
  // Create and deliver a buf for each packet (using net_rx()).
  //

  while (1) {
    int idx = (regs[E1000_RDT] + 1) % RX_RING_SIZE;

    if ((rx_ring[idx].status & E1000_RXD_STAT_DD) == 0) {
      return;
    }

    if (rx_ring[idx].status & E1000_RXD_STAT_EOP) {

      int len = rx_ring[idx].length;

      net_rx(rx_bufs[idx], len);
      rx_bufs[idx] = kalloc();
      if(rx_bufs[idx] == 0){
        return;
      }
      rx_ring[idx].status = 0;
      rx_ring[idx].addr = (uint64)rx_bufs[idx];
    }
    regs[E1000_RDT] = idx;
  }
}
```

e1000_transmit

```
#define E1000_TXD_CMD_RS 0x08000000 /* Report Status */
#define E1000_TXD_CMD_EOP 0x01000000 /* End of Packet */
```

To transmit a packet, the driver copies it into the next DMA descriptor in the transmit queue and informs the E1000 that another packet is available;

https://pdos.csail.mit.edu/6.1810/2024/readings/8254x_GBe_SDM.pdf

1. Finding the next packet index

   (by E1000_TDT control register)

2. Check overFlowed / not completed

3. Free the last buffer (kfree()) if there was

4. Fill the descriptor

```
int
e1000_transmit(char *buf, int len)
{
  // buf contains an ethernet frame; program it into
  // the TX descriptor ring so that the e1000 sends it. Stash
  // a pointer so that it can be freed after send completes.
  //
  // Acquire lock
  acquire(&e1000_lock);

  // get tail index
  int idx = regs[E1000_TDT];

  // check whether it is overflowed/not completed
  if ((tx_ring[idx].status & E1000_TXD_STAT_DD) == 0) {
    release(&e1000_lock);
    return -1;
  }

  // free the last buffer
  if (tx_bufs[idx])
    kfree(tx_bufs[idx]);

  tx_bufs[idx] = buf;
  tx_ring[idx].length = len;
  tx_ring[idx].addr = (uint64) (buf);
  tx_ring[idx].cmd = E1000_TXD_CMD_RS | E1000_TXD_CMD_EOP;
  tx_ring[idx].status = 0;

  // update ring location
  regs[E1000_TDT] = (idx + 1) % TX_RING_SIZE;

  release(&e1000_lock);
  return 0;
}
```

Labs
Part Two: UDP Receive

Problem Summary

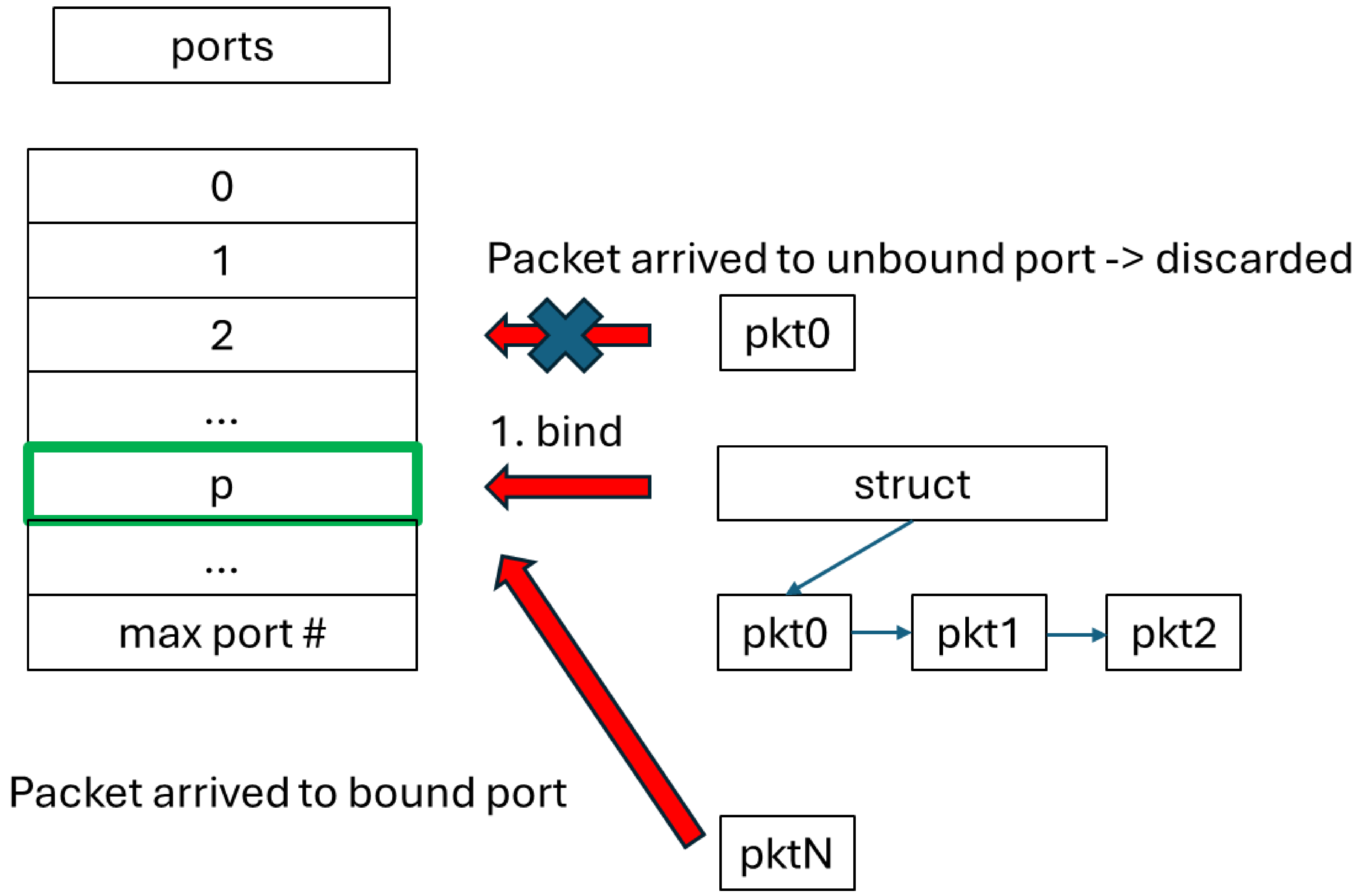
- A system call for **sending UDP packets** has already been implemented.
- The objective is to create a system call for **receiving UDP packets**

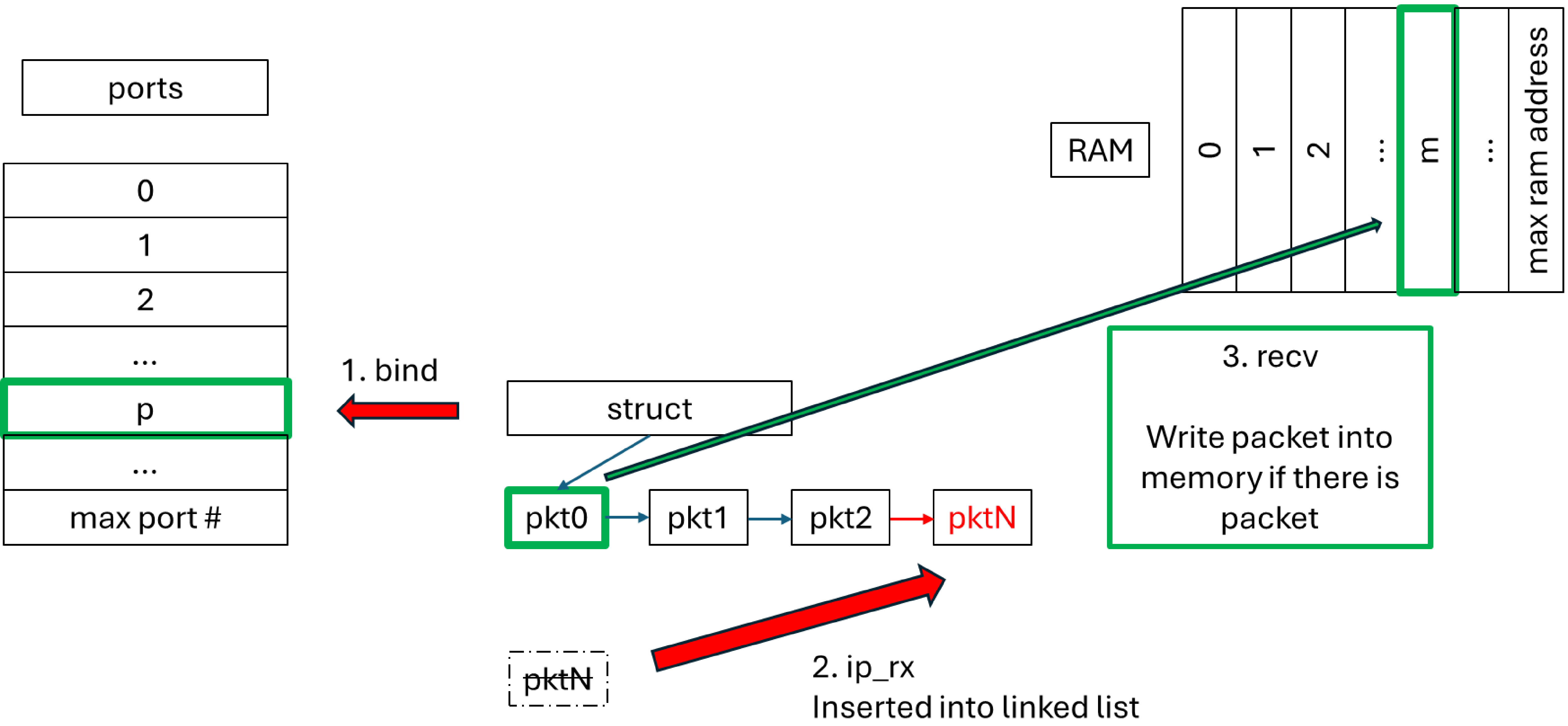
Related system-call API and function

- **`send(sport, dst, dport, *buf, len)`** (→ Already implemented)
  - Sends a packet to the destination specified by `(dst, dport)`.
  - **Packet:** Starts at `buf` and has a length of `len`.
- **`recv(dport, *src, *sport, *buf, len)`** (→ To be implemented)
  - Receives packets arriving at `dport`.
  - Writes the received information into `src` and `sprt`, which are virtual addresses.

- **`bind`** (→ To be implemented)
  - A process must bind to a port before calling `recv`.

- **`ip_rx`** (→ To be implemented)
  - Restores the packet before calling `recv`.


ports
0
1
2
...
p
...
max port #
Packet arrived to unbound port -> discarded
pkt0
1. bind
struct
pkt0
pkt1
pkt2
Packet arrived to bound port
pktN


ports
0
1
2
...
p
...
max port #
1. bind
struct
pkt0
pkt1
pkt2
pktN
~~pktN~~
2. ip_rx
Inserted into linked list
RAM
0
1
2
...
m
...
max ram address
3. recv
Write packet into memory if there is packet

bind(port)

- Binding **port** before calling **ip_rx** and **recv**
- The maximum number of packets that can arrive is limited to 16
- If a packet arrives at an unbound port, the packet should be discarded

Data structure

- Create "struct **bound_port**"
- Each bound port must store waiting packets.
- A list of **`struct bound_port`** instances is stored in the **`udb_table`.**
- The function **`find_bound_port`** is called to check whether the given port is bound or not.

```c
struct bound_port
{
  uint16 port;        // Bound port number
  int pending_count; // Number of pending packets
  struct spinlock lock;
  struct pending_packet *head; // Packet queue head
  struct pending_packet *tail; // Packet queue tail
};

#define MAX_PORTS 64
struct
{
  struct spinlock lock;
  struct bound_port ports[MAX_PORTS];
  int port_count;
} udp_table;

void
netinit(void)
{
  initlock(&netlock, "netlock"); // Initialize lock
  udp_table.port_count = 0;       // Initialize port counter
}
```

```c
static struct bound_port *
find_bound_port(uint16 port)
{
  for (int i = 0; i < udp_table.port_count; i++)
  {
    if (udp_table.ports[i].port == port)
      return &udp_table.ports[i]; // Return pointer to matching port structure
  }
  return 0; // Return 0 if not found
}
```

bind(port)

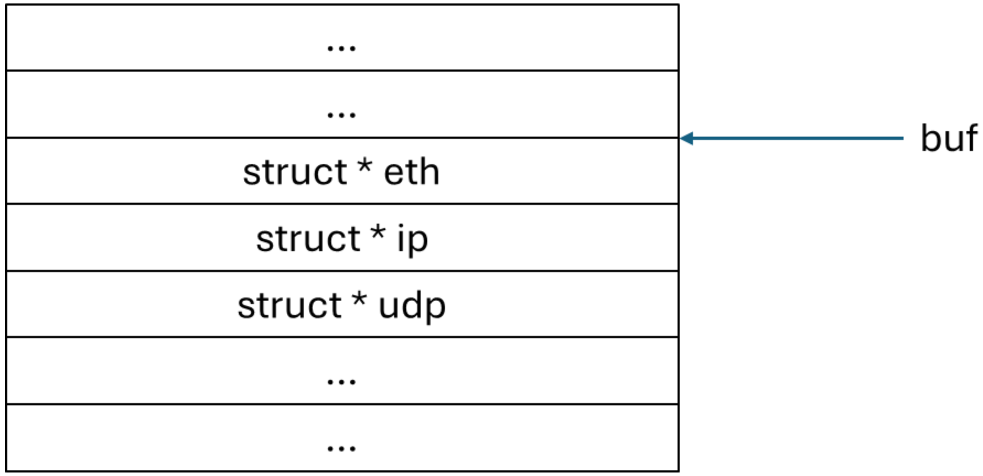
- A lock is necessary since other processes may try to bind the same port.

- If the port is already bound or the number of bound ports has already exce eded **MAX_PORTS**, the binding fails.

```c
uint64
sys_bind(void)
{
  int port; argint(0, &port);

  acquire(&udp_table.lock);

  {
    // Check if the port is already bound
    if (find_bound_port(port) != 0 || udp_table.port_count >= MAX_PORTS)
    {
      release(&udp_table.lock);
      return -1;
    }

    // Initialize a new bound_port structure
    struct bound_port *bp = &udp_table.ports[udp_table.port_count++];
    bp->port = port;
    bp->pending_count = 0;
    bp->head = bp->tail = 0;
    initlock(&bp->lock, "bound_port");
  }

  release(&udp_table.lock);
  return 0;
}
```

ip_rx(buf, len)

- This function must receive the packet and store it in the `queue` within `bound_port`.
- Starting from the `buf` address, the addresses of the **Ethernet packet**, **IP packet**, and **UDP packet** are stored.
- Packets are represented as structures in `net.h`.




```c
// a UDP packet header (comes after an IP header).
struct udp {
  uint16 sport; // source port
  uint16 dport; // destination port
  uint16 ulen;  // length, including udp header, not including IP header
  uint16 sum;   // checksum
};
```

Step1 of ip_rx(buf, len)

- Get the destination port from the UDP header.

- The function ntohs rearranges the byte order.
  - Packets use Big-endian
  - CPU use Little-endian

```c
void
ip_rx(char *buf, int len)
{
  // don't delete this printf; make grade depends on it.
  static int seen_ip = 0;
  if(seen_ip == 0)
    printf("ip_rx: received an IP packet\n");
  seen_ip = 1;

  struct eth *ethhdr = (struct eth *)buf;
  struct ip *iphdr = (struct ip *)(ethhdr + 1);
  struct udp *udphdr = (struct udp *)(iphdr + 1);
  uint16 dport = ntohs(udphdr->dport);

  acquire(&udp_table.lock);

  struct bound_port *bp = find_bound_port(dport);
  if (bp == 0)
  {
    kfree(buf);
    release(&udp_table.lock);
    return;
  }

  acquire(&bp->lock);
  release(&udp_table.lock);
```

```c
struct udp {
  uint16 sport; // source port
  uint16 dport; // destination port
  uint16 ulen;  // length, including udp header, not including IP header
  uint16 sum;   // checksum
};
```

Step2 of ip_rx(buf, len)

- If the number of waiting packets is equal to **MAX_PENDING_PACKETS**, return from the function.
- If not, initialize the packet.

```c
if (bp->pending_count >= MAX_PENDING_PACKETS)
{
  kfree(buf);
  release(&bp->lock);
  return;
}

int payload_len = ntohs(udphdr->ulen) - sizeof(struct udp);

char *data = kalloc();
struct pending_packet *pp = kalloc();

memmove(data, (char *)(udphdr + 1), payload_len);

memset(pp, 0, sizeof(*pp));
pp->data = data;
pp->len = payload_len;
pp->src_ip = ntohl(iphdr->ip_src);
pp->src_port = ntohs(udphdr->sport);
pp->next = 0;
```

Step3 of ip_rx(buf, len)

- Insert the prepared packet into the queue.

- It has to call the wakeup function.
  - I will explain later.

```c
// Update the queue
if (bp->tail == 0)
{
  bp->head = bp->tail = pp;
}
else
{
  bp->tail->next = pp;
  bp->tail = pp;
}
bp->pending_count++;

// Wake up the waiting process
wakeup(bp);
kfree(buf);
release(&bp->lock);
```

recv(dport, *src, *sport, *buf, len)

- Extract a packet stored in the queue.
- If no packet has arrived, wait until a new packet arrives.
- Store the IP address of the packet in `src` and store the port of the packet in `sport`.

Step1 of recv(dport, *src, *sport, *buf, len)

- Wait until the packet is received.
- If no packet has arrived, wait (by using sleep).
- It can be awakened by the previously mentioned wakeup.
    - The wakeup function is called if a packet arrives.
    - Exit the while loop and proceed to the next step.

```c
acquire(&udp_table.lock);
if ((bp = find_bound_port(dport)) == 0)
{
  release(&udp_table.lock);
  return -1;
}

acquire(&bp->lock);
release(&udp_table.lock);

while (bp->head == 0)
{
  if (p->killed)
  {
    release(&bp->lock);
    return -1;
  }
  sleep(bp, &bp->lock);
}
```

Step2 of recv(dport, *src, *sport, *buf, len)

- `Src` and `sport` are virtual addresses, which will store the source IP and source port, respectively.
- Extract the stored packet from the queue.
- Store the source IP and source port in the given virtual addresses.
    - Using copyout api

```c
pp = bp->head;
bp->head = pp->next;
if (bp->head == 0)
{
  bp->tail = 0;
}
bp->pending_count--;

int copy_len = pp->len;
if (copy_len > maxlen)
  copy_len = maxlen;

if (copyout(p->pagetable, src_arg, (char *)&pp->src_ip, sizeof(pp->src_ip)) < 0 ||
    copyout(p->pagetable, sport_arg, (char *)&pp->src_port, sizeof(pp->src_port)) < 0)
{
  goto bad;
}

if (copyout(p->pagetable, buf, pp->data, copy_len) < 0)
{
  goto bad;
}
```