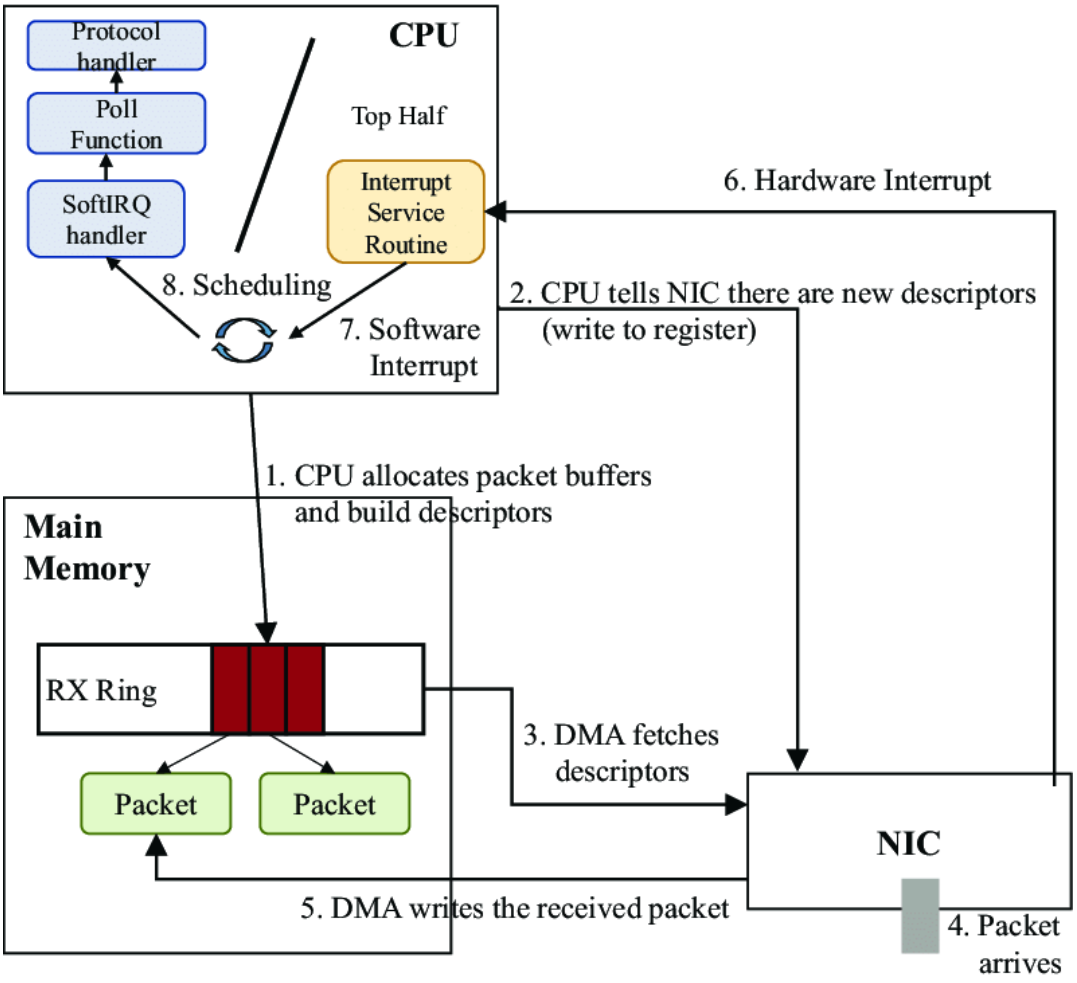
Lab6 networking

前置知识

网卡接收与传输



- 由这张图我们可以梳理下关于网卡收发包的细节，首先内核需要分配 `rx_ring` 和 `tx_ring` 两块环形缓冲区的内存用来接收和发送报文。其中以太网控制器的寄存器记录了关于 `rx_ring` 和 `tx_ring` 的详细信息。接收 packet 的细节如下：

    a. 内核首先在主存中分配内存缓冲区和环形缓冲区，并由 CPU 将 `rx_ring` 的详细信息写入以太网控制器

    b. 随后 NIC (Network Interface Card) 通过 DMA 获取到下一个可以写入的缓冲区的地址，当 packet 从硬件收到的时候外设通过 DMA 的方式写入对应的内存地址中

    c. 当写入内存地址后，硬件将会向 CPU 发生中断，操作系统检测到中断后会调用网卡的异常处理函数

    d. 异常处理函数可以通过由以太网控制寄存器映射到操作系统上的内存地址访问寄存器获取到下一个收到但未处理的 packet 的描述符，根据该描述符可以找到对应的缓冲区地址进行读取并传输给上层协议。

实验内容

Your Job(hard)

任务

- 你的任务是完成 `kernel/e1000.c` 文件中的 `e1000_transmit()` 和 `e1000_recv()` 函数，使驱动程序能够发送和接收数据包。目标是通过运行 `make grade` 来确保你的解决方案通过所有测试。

函数实现

```
int
e1000_transmit(struct mbuf *m)
{
  //
  // Your code here.
  //
  // the mbuf 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 sending.
  //
```

```
  // 获取 ring position
  acquire(&e1000_lock);

  uint64 tdt = regs[E1000_TDT];
  uint64 index = tdt % TX_RING_SIZE;
  struct tx_desc send_desc = tx_ring[index];
  if(!(send_desc.status & E1000_TXD_STAT_DD)) {
    release(&e1000_lock);
    return -1;
  }

  if(tx_mbufs[index] != 0){
    // 如果该位置的缓冲区不为空则释放
    mbuffree(tx_mbufs[index]);
  }

  tx_mbufs[index] = m;
  tx_ring[index].addr = (uint64)tx_mbufs[index]->head;
  tx_ring[index].length = (uint16)tx_mbufs[index]->len;
  tx_ring[index].cmd = E1000_TXD_CMD_RS | E1000_TXD_CMD_EOP;
  tx_ring[index].status = 0;

  tdt = (tdt + 1) % TX_RING_SIZE;
  regs[E1000_TDT] = tdt;
  __sync_synchronize();

  release(&e1000_lock);
  return 0;
}

static void
e1000_recv(void)
{
  //
  // Your code here.
  //
  // Check for packets that have arrived from the e1000
  // Create and deliver an mbuf for each packet (using
net_rx()).
  //
  // 获取接收 packet 的位置
  uint64 rdt = regs[E1000_RDT];
```

```
  uint64 index = (rdt + 1) % RX_RING_SIZE;

  // acquire(&e1000_lock);  // 锁定以进行安全的并发访问

  if(!(rx_ring[index].status & E1000_RXD_STAT_DD)){
    // 查看新的 packet 是否有 E1000_RXD_STAT_DD 标志，如果没有，则直接返回
    return;
  }
  while(rx_ring[index].status & E1000_RXD_STAT_DD){
    // 使用 mbufput 更新长度并将其交给 net_rx() 处理
    struct mbuf* buf = rx_mbufs[index];
    mbufput(buf, rx_ring[index].length);

    // 分配新的 mbuf 并将其写入到描述符中并将状态码设置成 0
    rx_mbufs[index] = mbufalloc(0);
    rx_ring[index].addr = (uint64)rx_mbufs[index]->head;
    rx_ring[index].status = 0;
    rdt = index;
    regs[E1000_RDT] = rdt;
    __sync_synchronize();

    // 将数据包传递给net_rx()处理
    net_rx(buf);

    // 更新index，继续处理下一个接收到的数据包
    index = (regs[E1000_RDT] + 1) % RX_RING_SIZE;
  }

  // release(&e1000_lock);  // 在循环结束后释放锁
}
```

`socket`实现过程

- 在类 Unix 操作系统上面，设备、`pipe` 和 `socket` 都要当做文件来处理，但在操作系统处理的时候需要根据文件描述符来判断是什么类型的文件并对其进行分发给不同的部分进行出来，我们需要实现的就是操作系统对于 `socket` 的处理过程。

  `socket` 的读取过程需要根绝给定的 `socket` 从所有 `sockets` 中找到并读取 `mbuf`，当对应的 `socket` 的缓冲区为空的时候则需要进行 `sleep` 从而将 CPU 时间让给调度器，当对应的 `socket` 收到了 packet 的时候再唤醒对应的进程:

```
sockrecvudp(struct mbuf *m, uint32 raddr, uint16 lport, uint16
rport)
{
  //
  // Your code here.
  //
  // Find the socket that handles this mbuf and deliver it,
waking
  // any sleeping reader. Free the mbuf if there are no sockets
  // registered to handle it.
  //
  acquire(&lock);
  struct sock* sock = sockets;
  // 首先找到对应的 socket
  while(sock != 0){
    if(sock->lport == lport && sock->raddr == raddr && sock-
>rport == rport){
      break;
    }
    sock = sock->next;
    if(sock == 0){
      printf("[Kernel] sockrecvudp: can't find socket.\n");
      return;
    }
  }
  release(&lock);
  acquire(&sock->lock);
  // 将 mbuf 分发到 socket 中
  mbufq_pushtail(&sock->rxq, m);
  // 唤醒可能休眠的 socket
  release(&sock->lock);
  wakeup((void*)sock);
}

int sock_read(struct sock* sock, uint64 addr, int n){
  acquire(&sock->lock);
  while(mbufq_empty(&sock->rxq)) {
    // 当队列为空的时候，进入 sleep，将 CPU
    // 交给调度器
    if(myproc()->killed) {
      release(&sock->lock);
      return -1;
```

```
    }
    sleep((void*)sock, &sock->lock);
  }
  int size = 0;
  if(!mbufq_empty(&sock->rxq)){
    struct mbuf* recv_buf = mbufq_pophead(&sock->rxq);
    if(recv_buf->len < n){
      size = recv_buf->len;
    }else{
      size = n;
    }
    if(copyout(myproc()->pagetable, addr, recv_buf->head, size)
!= 0){
      release(&sock->lock);
      return -1;
    }
    // 或许要考虑一下读取的大小再考虑是否释放，因为有可能
    // 读取的字节数要比 buf 中的字节数少
    mbuffree(recv_buf);
  }
  release(&sock->lock);
  return size;
}
```

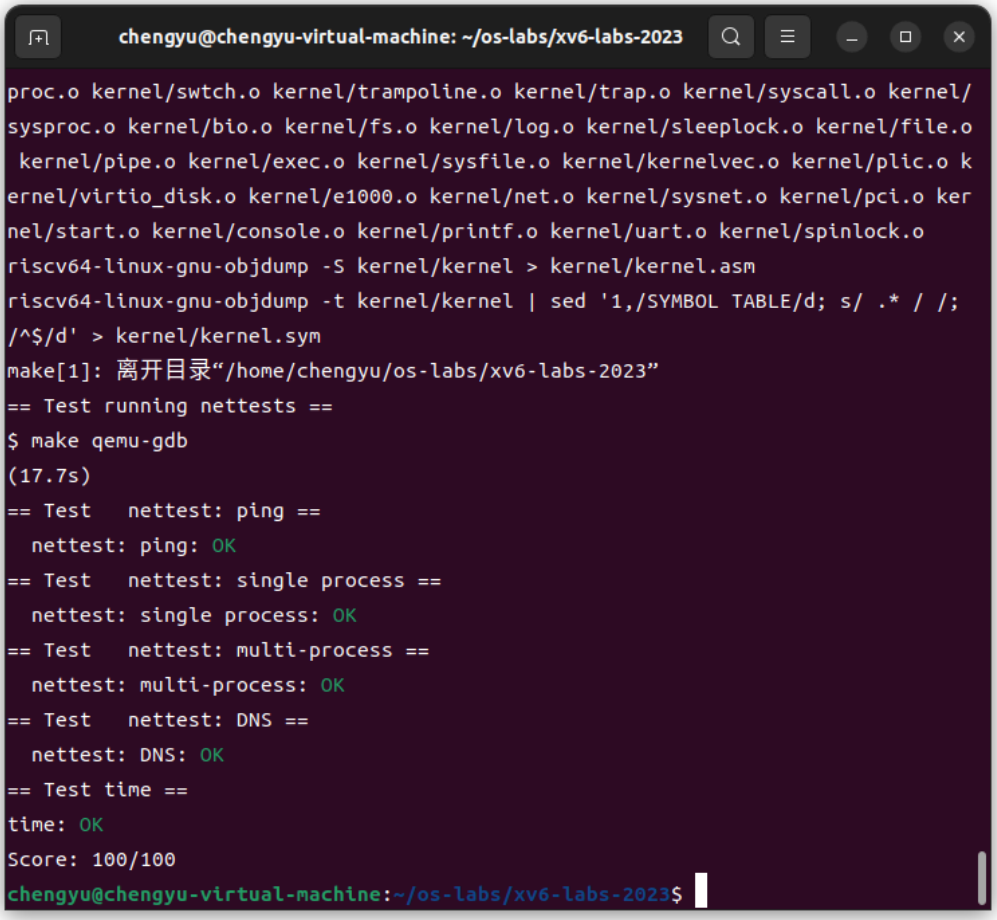
- 写`socket`的过程：

```
int sock_write(struct sock* sock, uint64 addr, int n){
  acquire(&sock->lock);
  struct mbuf* send_buf = mbufalloc(sizeof(struct udp) +
sizeof(struct ip) + sizeof(struct eth));
  if (copyin(myproc()->pagetable, (char*)send_buf->head, addr,
n) != 0){
    release(&sock->lock);
    return -1;
  }
  mbufput(send_buf, n);
  net_tx_udp(send_buf, sock->raddr, sock->lport, sock->rport);
  release(&sock->lock);
  return n;
}
```

- 关闭`socket`的过程：

```
void sock_close(struct sock* sock){
  struct sock* prev = 0;
  struct sock* cur = 0;
  acquire(&lock);
  // 遍历 sockets 链表找到对应的 socket 并将其
  // 从链表中移除
  cur = sockets;
  while(cur != 0){
    if(cur->lport == sock->lport && cur->raddr == sock->raddr &&
cur->rport == sock->rport){
      if(cur == sockets){
        sockets = sockets->next;
        break;
      }else{
        sock = cur;
        prev->next = cur->next;
        break;
      }
    }
    prev = cur;
    cur = cur->next;
  }
  // 释放 sock 所有的 mbuf
  acquire(&sock->lock);
  while(!mbufq_empty(&sock->rxq)){
    struct mbuf* free_mbuf = mbufq_pophead(&sock->rxq);
    mbuffree(free_mbuf);
  }
  // 释放 socket
  release(&sock->lock);
  release(&lock);
  kfree((void*)sock);
}
```

实验得分


chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
proc.o kernel/swtch.o kernel/trampoline.o kernel/trap.o kernel/syscall.o kernel/
sysproc.o kernel/bio.o kernel/fs.o kernel/log.o kernel/sleeplock.o kernel/file.o
kernel/pipe.o kernel/exec.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o k
ernel/virtio_disk.o kernel/e1000.o kernel/net.o kernel/sysnet.o kernel/pci.o ker
nel/start.o kernel/console.o kernel/printf.o kernel/uart.o kernel/spinlock.o
riscv64-linux-gnu-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /;
/^$/d' > kernel/kernel.sym
make[1]: 离开目录“/home/chengyu/os-labs/xv6-labs-2023”
== Test running nettests ==
$ make qemu-gdb
(17.7s)
== Test   nettest: ping ==
  nettest: ping: OK
== Test   nettest: single process ==
  nettest: single process: OK
== Test   nettest: multi-process ==
  nettest: multi-process: OK
== Test   nettest: DNS ==
  nettest: DNS: OK
== Test time ==
time: OK
Score: 100/100
chengyu@chengyu-virtual-machine:~/os-labs/xv6-labs-2023$