Internet device driver adv os final

Student: Suwei Yang UID:u1429034

1. Explain the problem or question that the group explored in a paragraph or two;

Before starting to work on this project, there are two things to do: read the E1000 manual and the driver code. From the manual, we can understand roughly the driver's data structure, like descriptor, command, status…etc., and the driver's transmitting and receiving flow.

Afterward, we have the basic knowledge to the code. We start from the driver code. There are two components to be finished, e1000_transmit() and e1000_recv(). We can complete them via the explanation on MIT website.

2. Explain the key aspects of the work you did;

To make the driver act, I need to finish two components left by MIT team: e1000_transmit() and e1000_recv(). They are used to set hardware before transmitting and after receiving message to/from server.

```
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.
  //
  //printf("Suwei: something need to send here.\n");
  acquire(&e1000_lock);
  uint32 transit_tail = regs[E1000_TDT];//get availiable index
  if((tx_ring[transit_tail].status & E1000_TXD_STAT_DD)==0)//check if availiable tx desc is ready.
  {
    release(&e1000_lock);
    return -1;//return errro, because not ready
  }
  if(tx_mbufs[transit_tail]!=0)
    mbuffree(tx_mbufs[transit_tail]);//free mbufer from avaliable tx descriptor
  tx_ring[transit_tail].addr = (uint64)m->head;
  tx_ring[transit_tail].length = m->len;
  tx_ring[transit_tail].cmd = E1000_TXD_CMD_EOP | E1000_TXD_CMD_RS;//set cmd to hw
  tx_mbufs[transit_tail] = m;
  regs[E1000_TDT] = (transit_tail + 1)%TX_RING_SIZE;//add tail
  release(&e1000_lock);
  return 0;
}
```

From e1000_transmit(), we will get a buffer, mbuf, containing a couple of things: message length, content, IP, and port. We need to put information into corresponding registers and then issue a command afterward. Hardware will push the data to the server we specified. Finally, we need to increase the tail by 1 to tell the hardware the next available space for the next transmission.

```c
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()).
  //
  //printf("Suwei: something need to receive here.\n");
  uint32 receive_tail = (regs[E1000_RDT]+ 1)% RX_RING_SIZE;//get received tail
  struct rx_desc *rx_des= &rx_ring[receive_tail];//get received descriptor
  struct mbuf *m = rx_mbufs[receive_tail];
  while(rx_des->status& E1000_RXD_STAT_DD)
  {
    m->len = rx_des->length;
    net_rx(m);
    m = mbufalloc(0);
    rx_mbufs[receive_tail] = m;//update new sapce for mbuf
    rx_des->addr=(uint64)m->head;
    rx_des->status = 0;//clear status bits
    receive_tail = (receive_tail+1)%RX_RING_SIZE;//add tail
    rx_des= &rx_ring[receive_tail];
    m = rx_mbufs[receive_tail];
    //need loops in this function to check the tail.
  }
  regs[E1000_RDT] = (receive_tail-1)%RX_RING_SIZE;
}
```

In e1000_rec(), we will need to handle the flow after receiving the message from server. Only two things that need to be finished: putting mbuf into net_rx() and clearing space for following received package. In net_rx(), driver will find a corresponding socket for received package, as shown below. Then, clear move can have a space for upcoming message.

```c
void
sockrecvudp(struct mbuf *m, uint32 raddr, uint16 lport, uint16 rport)
{
  //
  // 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.
  //
  struct sock *si;

  acquire(&lock);
  si = sockets;
  while (si) {
    if (si->raddr == raddr && si->lport == lport && si->rport == rport)
      goto found;
    si = si->next;
  }
  release(&lock);
  mbuffree(m);
  return;

found:
  acquire(&si->lock);
  mbufq_pushtail(&si->rxq, m);
  wakeup(&si->rxq);
  release(&si->lock);
  release(&lock);
}
```

After finishing the two components, the driver can pass the test script, and we can start doing some improvement. I found the they used copyin()/copyout() as the communication between user and kernel, which might be the bottleneck for the driver. Hence I created another function, mbufalloc_suwei, to map mbuf into user memory space (shown below). Hoping to improve the performance by reducing memory movement during the kernel.

```c
int
sockwrite(struct sock *si, uint64 addr, int n)
{
  struct proc *pr = myproc();//suwei comment
  struct mbuf *m;

  m = mbufalloc(MBUF_DEFAULT_HEADROOM);//suwei comment
  //m = mbufalloc_suwei(MBUF_DEFAULT_HEADROOM,addr, n);//suwei add
  //mbufput(m, n);//suwei add
  if (!m)
    return -1;
  #if 1//suwei add
  if (copyin(pr->pagetable, mbufput(m, n), addr, n) == -1) {
    mbuffree(m);
    return -1;
  }
  #endif
  net_tx_udp(m, si->raddr, si->lport, si->rport);
  return n;
}
```

```c
//Allocates a packet buff, suwei add
//use user pa as driver space.
struct mbuf *
mbufalloc_suwei(unsigned int headroom,uint64 addr, int n)
{
  struct proc *pr = myproc();
  pagetable_t pagetable = pr->pagetable;
  uint64 va0, pa0;
  struct mbuf *m;
  va0 = PGROUNDUP(addr);
  pa0 = walkaddr(pagetable, va0);//va is continuous, but pa is continuous in a 4K
  //memset((void*)pa0, 0, PGSIZE);//suwei add
  if (headroom > MBUF_SIZE)
    return 0;
  //m = kalloc();//suwei comment
  m = (struct mbuf *)pa0;//suwei add
  if (m == 0)
    return 0;
  m->next = 0;
  m->head = (char *)m->buf + headroom;
  m->len = 0;
  //memset(m->buf, 0, sizeof(m->buf));//suwei comment
  return m;
}
```

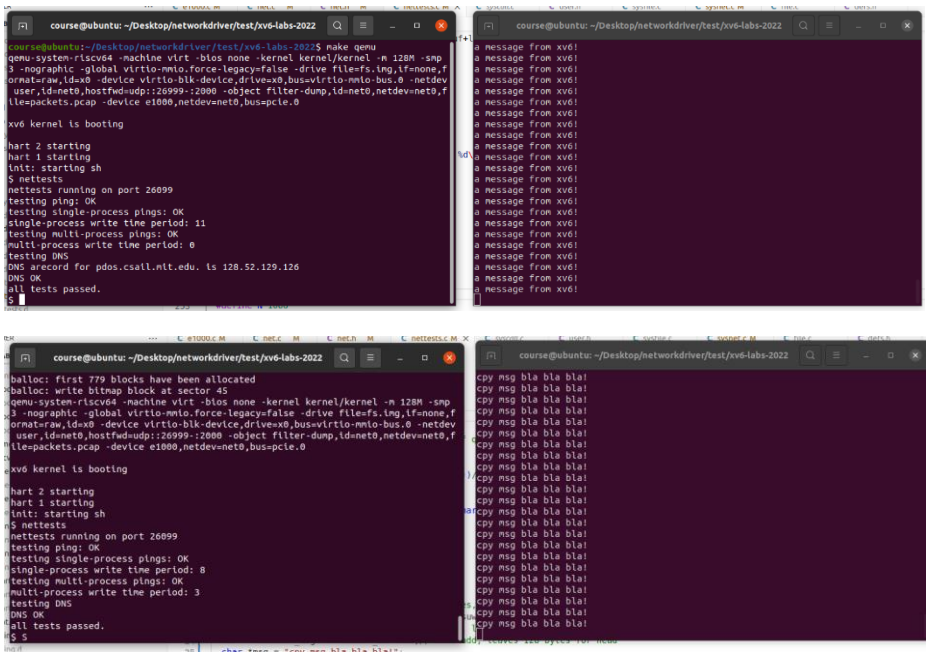
3. Explain what was interesting/exciting/promising about what you did;

The most exciting part for me was when the driver sent package to server and when performance got different after I modified it. It still has some improved space like read flow. We can create buffer space for the message and map it directly to user space to further reduce the memory operation. However, it might take more time to implement and debug.

4. Explain what limitations you ran into in the work; what questions couldn't you answer; what prevented you from answering them;

From the result shown below, single process test improved after modification. I used a longer message to test afterward and it was 11ms(improved) v.s. 29ms(original). However, multi-process test elapsed time increased after the improvement. I didn’t figure out what they were waiting for. Especially when I increased the message length, they increased further. I guess it is probably related to the job in user space.

5. Result



Single process/ uptime()	T1	T2	T3	T4	T5
Original	11	12	13	15	14
Improved	8	9	15	11	15