# Lab5

## Problem 1: NIC

### `e1000_transmit`(发送数据包)

**代码如下**:

```c
int
e1000_transmit(char *buf, int len)
{
  //
  // Your code here.
  //
  // 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.
  //

  // Add lock
  acquire(&e1000_lock);
  // Find the next available descriptor
  uint32 idx = regs[E1000_TDT];
  // Check if the descriptor is free
  if (!(tx_ring[idx].status & E1000_TXD_STAT_DD)) {
    // TX ring full
    release(&e1000_lock);
    return -1;
  }
  // Free the previous buffer if it exists
  if (tx_bufs[idx]) {
    kfree(tx_bufs[idx]);
  }
  // Use the provided buffer directly
  tx_ring[idx].addr = (uint64)buf;
  tx_ring[idx].length = len;
  tx_ring[idx].cmd = E1000_TXD_CMD_EOP | E1000_TXD_CMD_RS;
  tx_ring[idx].status = 0; // Clear status to indicate it's in use
  tx_bufs[idx] = buf; // Stash the buffer pointer
  // Update the TDT to point to the next descriptor
  regs[E1000_TDT] = (idx + 1) % TX_RING_SIZE;
  release(&e1000_lock);

  return 0;
}
```

**逻辑思路**:

1. 首先获取 `e1000_lock` 锁，确保对发送描述符环的操作是线程安全的。
2. 获取当前的发送索引 `TDT`, 读取`E1000_TXD_STAT_DD`寄存器.
3. 检查发送队列是否已满。如果当前描述符的状态没有设置 `DD` 标志，说明描述符还在使用中，返回错误。
4. 如果当前描述符已经使用过，释放之前的缓冲区。
5. 填充发送描述符，包括地址、长度、命令、状态和缓冲区。

6. 更新 `TDT` 寄存器，通知硬件.
7. 释放锁，返回成功。

### `e1000_recv`(接收数据包)

这个函数功能主要是被`e1000_intr`调用,用于检查接受环`rx_ring`中是否有新的数据包到达,如果有则将数据包从网卡拷贝到用户提供的缓冲区中.

**代码如下**:

```c
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()).
  //

  uint32 idx = (regs[E1000_RDT] + 1) % RX_RING_SIZE;
  // Loop through received packets
  while (rx_ring[idx].status & E1000_RXD_STAT_DD) {
    // Packet received
    int len = rx_ring[idx].length;
    char *buf = kalloc();
    if (!buf) {
      panic("e1000_recv: kalloc failed");
    }
    memmove(buf, rx_bufs[idx], len);
    net_rx(buf, len);
    // Replenish the buffer
    rx_ring[idx].addr = (uint64)rx_bufs[idx];
    rx_ring[idx].status = 0; // Clear status to indicate it's free
    // Update RDT to point to the next descriptor
    regs[E1000_RDT] = idx;
    idx = (idx + 1) % RX_RING_SIZE;
  }

}
```

**逻辑思路**:

1. 计算起始位置 `RDT + 1` ，因为 `RDT` 指向最后一个已处理的描述符。
2. 循环检查描述符的状态，如果 `DD` 标志被设置，说明有新的数据包到达。
3. 为每个接收到的数据包分配一个新的缓冲区，并将数据从接收缓冲区复制到新缓冲区。
4. 调用 `net_rx` 函数将数据包传递给*上层网络协议栈*进行处理。
5. 重新填充接收描述符，设置地址并清除状态。
6. 更新 `RDT` 寄存器，通知硬件已处理的数据包。
7. 重复上述步骤，直到没有新的数据包为止。

---

## Problem 2: UDP Receive

`UDP 网络栈`需要:

- **端口到数据包的映射**: 当数据包到达某个端口时,需要知道把他放在哪个队列中.
- **有限的长度的队列**: 避免内存耗尽.
- **监听多个端口**: 支持同时不同进程监听不同端口.

## 数据结构设计

**代码如下**:

```c
// Maximum number of packets that can be queued for a port
#define MAX_QUEUE_SIZE 32
// Maximum number of ports that can be bound
#define MAX_PORTS 16

// The struct of data packet
struct packet {
  char *buf;            // packet buffer
  int len;              // packet length
  uint32 src_ip;        // source IP address (host byte order)
  uint16 src_port;      // source port (host byte order)
  struct packet *next; // next packet in queue
};

// A port that has been bound
struct port_entry {
  int port;             // port number (host byte order), 0 means unused
  struct packet *head; // head of packet queue
  struct packet *tail; // tail of packet queue
  int queue_size;       // current queue size
};

static struct port_entry port_table[MAX_PORTS];
```

## `sys_bind()` (绑定端口)

**思路**:

这个其实就是根据我们自己设计的数据结构,实现端口号和端口队列的映射关系,进行初始化,实现在队列中插入数据包.

**代码如下**:

```c
uint64
sys_bind(void)
{
  int port;

  argint(0, &port);

  acquire(&netlock);

  // Check if port is already bound
  for(int i = 0; i < MAX_PORTS; i++){
    if(port_table[i].port == port){
```

```c
      // Port already bound, clean up old queue
      struct packet *pkt = port_table[i].head;
      while(pkt){
        struct packet *next = pkt->next;
        kfree(pkt->buf);
        kfree((char *)pkt);
        pkt = next;
      }
      port_table[i].head = 0;
      port_table[i].tail = 0;
      port_table[i].queue_size = 0;
      release(&netlock);
      return 0;
    }
  }

  // Find an empty slot
  for(int i = 0; i < MAX_PORTS; i++){
    if(port_table[i].port == 0){
      port_table[i].port = port;
      port_table[i].head = 0;
      port_table[i].tail = 0;
      port_table[i].queue_size = 0;
      release(&netlock);
      return 0;
    }
  }

  release(&netlock);
  return -1; // no free slots
}
```

## `ip_rx()` (接受IP数据包)

**思路**:

当网卡接收到数据包的时候,就会调用这个函数,因此这个函数需要:

- 检查这个是否为`UDP包`
- 提取端口号
- 找到对应的端口队列
- 将数据包插入
- 唤醒等待的进程

> **Note**
>
> 注意`大端`和`小端`的转换以及`wakeup()`和`sleep()`的配合

**代码如下**:

```c
void
ip_rx(char *buf, int len)
{
  // don't delete this printf; make grade depends on it.
```

```c
static int seen_ip = 0;
if(seen_ip == 0)
  printf("ip_rx: received an IP packet\n");
seen_ip = 1;

struct eth *eth = (struct eth *)buf;
struct ip *ip = (struct ip *)(eth + 1);

// Check if it's a UDP packet
if(ip->ip_p != IPPROTO_UDP){
  kfree(buf);
  return;
}

// Check packet length
int ip_len = ntohs(ip->ip_len);
if(len < sizeof(struct eth) + ip_len){
  kfree(buf);
  return;
}

struct udp *udp = (struct udp *)(ip + 1);
int dport = ntohs(udp->dport);  // destination port (host byte order)
int sport = ntohs(udp->sport);  // source port (host byte order)
uint32 src_ip = ntohl(ip->ip_src);  // source IP (host byte order)

acquire(&netlock);

// Find the port entry
struct port_entry *entry = 0;
for(int i = 0; i < MAX_PORTS; i++){
  if(port_table[i].port == dport){
    entry = &port_table[i];
    break;
  }
}

// If port not bound or queue is full, drop the packet
if(entry == 0 || entry->queue_size >= MAX_QUEUE_SIZE){
  release(&netlock);
  kfree(buf);
  return;
}

// Create a new packet entry
struct packet *pkt = (struct packet *)kalloc();
if(pkt == 0){
  release(&netlock);
  kfree(buf);
  return;
}

pkt->buf = buf;
pkt->len = len;
pkt->src_ip = src_ip;
pkt->src_port = sport;
pkt->next = 0;
```

```c
  // Add to queue
  if(entry->tail == 0){
    entry->head = pkt;
    entry->tail = pkt;
  } else {
    entry->tail->next = pkt;
    entry->tail = pkt;
  }
  entry->queue_size++;

  // Wake up any process waiting for packets on this port
  wakeup(&entry->head);

  release(&netlock);
}
```

### `sys_recv()`(接收数据包)

**思路**: 这个系统调用应该是要从队列取出数据包,并把数据包内容拷贝到用户态缓冲区中.因此需要:

- 如果队列为空,则睡眠等待
- 取出队列头的数据包
- 拷贝数据包`payload`到用户缓冲区
- 释放数据包内存

**代码如下**:

```c
uint64
sys_recv(void)
{
  int dport;
  uint64 src_addr;  // user pointer to int
  uint64 sport_addr; // user pointer to short
  uint64 buf_addr;   // user pointer to char[]
  int maxlen;

  argint(0, &dport);
  argaddr(1, &src_addr);
  argaddr(2, &sport_addr);
  argaddr(3, &buf_addr);
  argint(4, &maxlen);

  struct proc *p = myproc();

  acquire(&netlock);

  // Find the port entry
  struct port_entry *entry = 0;
  for(int i = 0; i < MAX_PORTS; i++){
    if(port_table[i].port == dport){
      entry = &port_table[i];
      break;
    }
  }

  // Port not bound
```

```
  if(entry == 0){
    release(&netlock);
    return -1;
  }

  // Wait for a packet if queue is empty
  while(entry->head == 0){
    sleep(&entry->head, &netlock);
  }

  // Dequeue the first packet
  struct packet *pkt = entry->head;
  entry->head = pkt->next;
  if(entry->head == 0){
    entry->tail = 0;
  }
  entry->queue_size--;

  release(&netlock);

  // Extract packet data
  struct eth *eth = (struct eth *)pkt->buf;
  struct ip *ip = (struct ip *)(eth + 1);
  struct udp *udp = (struct udp *)(ip + 1);
  char *payload = (char *)(udp + 1);

  // Calculate payload length
  int udp_len = ntohs(udp->ulen);
  int payload_len = udp_len - sizeof(struct udp);

  // Copy up to maxlen bytes
  int copy_len = payload_len < maxlen ? payload_len : maxlen;

  // Copy source IP to user space (host byte order)
  if(copyout(p->pagetable, src_addr, (char *)&pkt->src_ip, sizeof(uint32)) < 0){
    kfree(pkt->buf);
    kfree((char *)pkt);
    return -1;
  }

  // Copy source port to user space (host byte order)
  if(copyout(p->pagetable, sport_addr, (char *)&pkt->src_port, sizeof(uint16)) < 0){
    kfree(pkt->buf);
    kfree((char *)pkt);
    return -1;
  }

  // Copy payload to user space
  if(copyout(p->pagetable, buf_addr, payload, copy_len) < 0){
    kfree(pkt->buf);
    kfree((char *)pkt);
    return -1;
  }

  // Free the packet buffer and packet structure
  kfree(pkt->buf);
  kfree((char *)pkt);
```

```
    return copy_len;
}
```

---

## 总结

这次的lab做下来,感觉就是又发现自己有没接触过的东西,然后就是文档套文档的了解,然后进行一个结合 `AI` 的一个理解加上实现.

通过AI一个辅助理解可以知道:

| 层级 | 关键文件 | 作用与交互逻辑 |
|---|---|---|
| 0. 模拟硬件 | `Makefile` | 配置者。告诉 QEMU：启用网络 ( `-netdev user` )，模拟 E1000 网卡 ( `-device e1000` )，并设置端口转发。 |
| 1. 总线驱动 | `kernel/pci.c` | 发现者。系统启动时扫描 PCI 总线。当它发现 ID 为 `0x100e8086` 的设备时，它做了一件关键的事：告诉网卡“以后我读写 `0x40000000` 这个内存地址，就是读写你的寄存器” (MMIO)。 |
| 2. 设备定义 | `kernel/e1000_dev.h` | 字典。定义了 E1000 硬件的所有寄存器偏移量（如 `E1000_TDT` ）和标志位（如 `E1000_TXD_STAT_DD` ）。你需要频繁查阅这个文件来理解寄存器的含义。 |
| 3. 网卡驱动 | `kernel/e1000.c` | 执行者 (你的战场)。这里维护了 `RX Ring` (接收环) 和 `TX Ring` (发送环)。交互上：向下通过 DMA 和中断与 QEMU 交互；向上提供 `e1000_transmit` 和 `e1000_recv` 给协议栈调用。 |
| 4. 网络协议栈 | `kernel/net.c` | 翻译官。处理 IP、UDP、ARP 协议头。它不关心怎么发出去，只调用 `e1000_transmit`。它也不关心怎么收进来，只等待 `e1000_recv` 把解析好的 `mbuf` 扔给它。 |
| 5. 系统调用 | `kernel/sysfile.c` | 接口。实现了 `socket`, `bind`, `connect` 等系统调用，把用户态的数据搬运到内核态的 `mbuf` 中。 |

之前对于 `xv6` 的学习,我是没有解除网络这个方面的,这次算是补全了这个知识点.

之前是想着自己就是了解一些协议,比如 `TCP`, `UDP`, `ARP`, `IP` 这些协议的基本原理,然后知道通过 `qemu` 模拟网卡硬件来实现网络通信. 但是网络这个入门也是有点难度的,毕竟涉及的东西比较多且杂.通过这次的实验,也算是第一次上手系统级的网络编程.

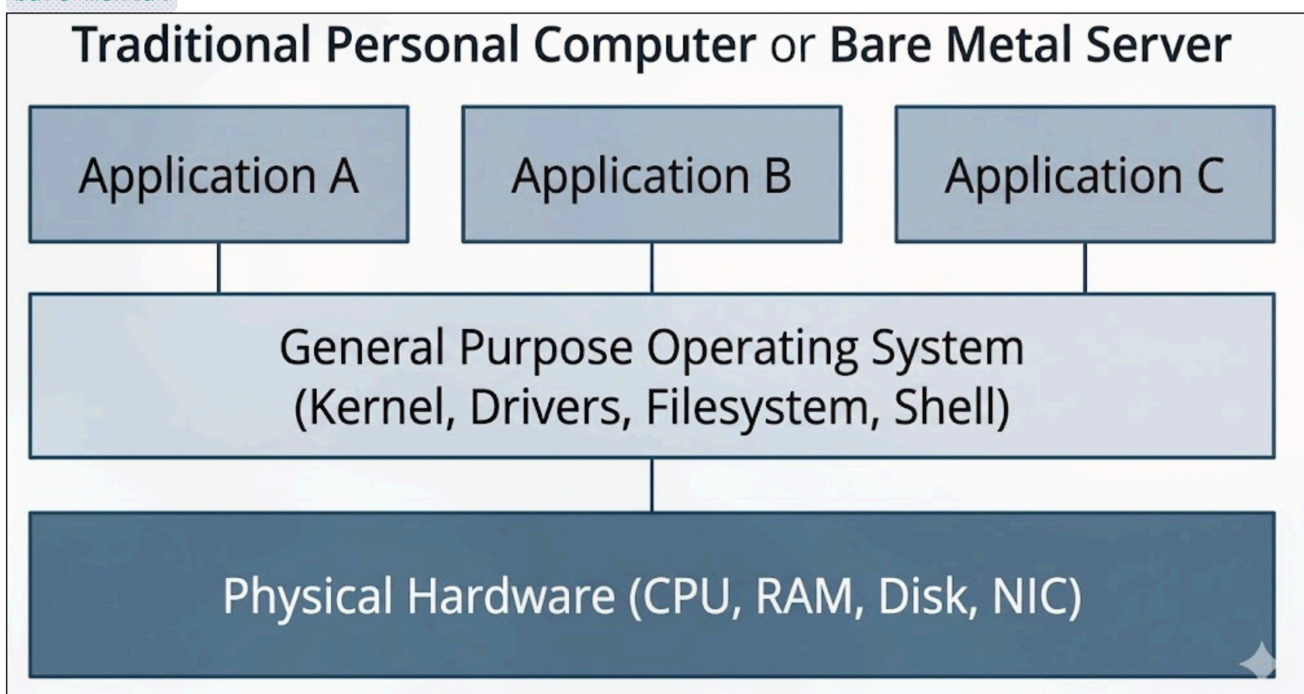
这次的实验,不仅仅是最后一次实验,也是这个课程的结束,这个课程和 `xv6` 一样,小而精. 通过一个个实验,让我真正上手实操操作系统的方方面面,从最开始的系统调用,到系统攻击,到address space,再到文件系统,最后到网络. 这也让我再一次体会到系统的内核编写的复杂性和趣味性,自己个人模仿从零开始一个编写系统内核还是有点难度的😢,不知道能不能在大学毕业前,写出一个属于自己的小型操作系统内核🤔.

最近也了解到一个新的内容`Unikernel`,它是一种将应用程序和操作系统内核紧密结合在一起的计算范式,算是**操作系统虚拟化**的一个方向:**容器化**的一个再升级.

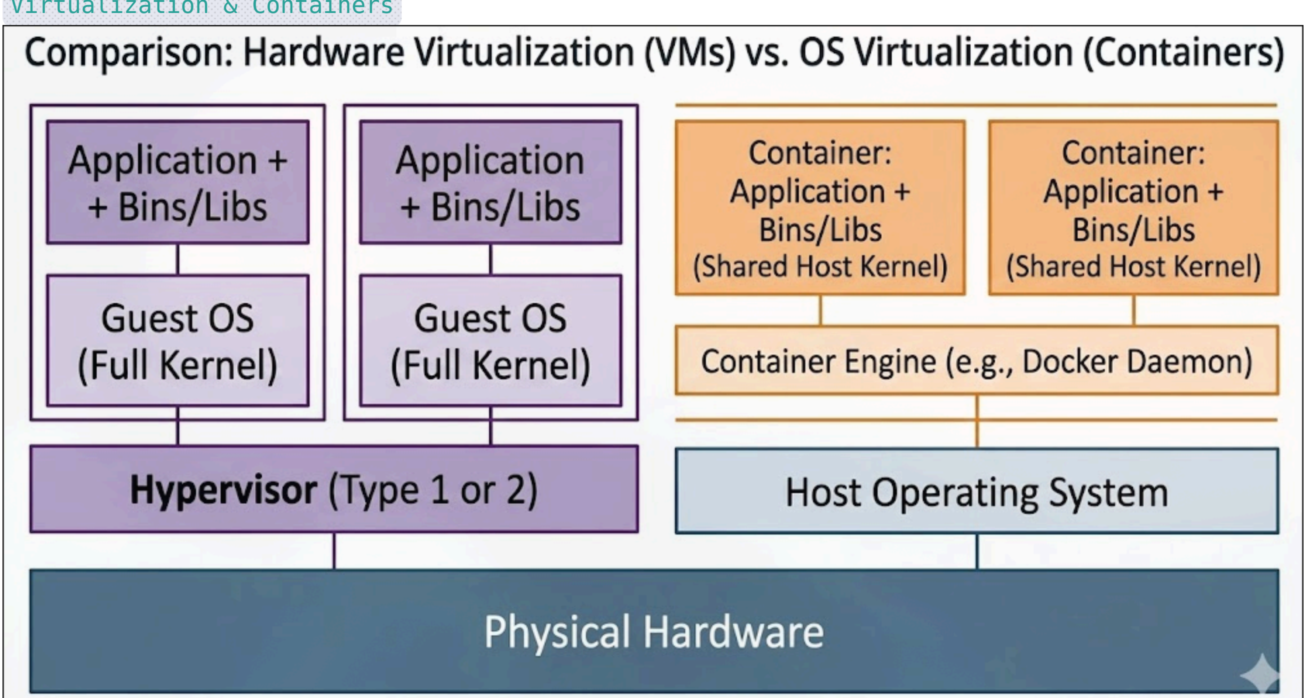
近代操作系统经历的是:

- **宏内核**(Monolithic Kernel): 传统的操作系统设计,内核包含了所有的服务和驱动程序,如`Linux`,`Unix`.
- **微内核**(Microkernel): 将内核功能最小化,只保留最基本的服务,如进程管理和通信,其他服务运行在用户空间,如`Minix`,`QNX`.
- **虚拟化**:
  - **虚拟机**(VMs): 每个虚拟机运行一个完整的操作系统,如`VMware`,`VirtualBox`.
  - **容器**: 共享主机操作系统内核,但隔离应用程序环境,如`Docker`,`Kubernetes`.
- **Unikernel**: 将应用程序和操作系统内核合并为一个单一的可执行映像,只包含应用程序运行所需的最小内核功能,如`MirageOS`,`IncludeOS`.

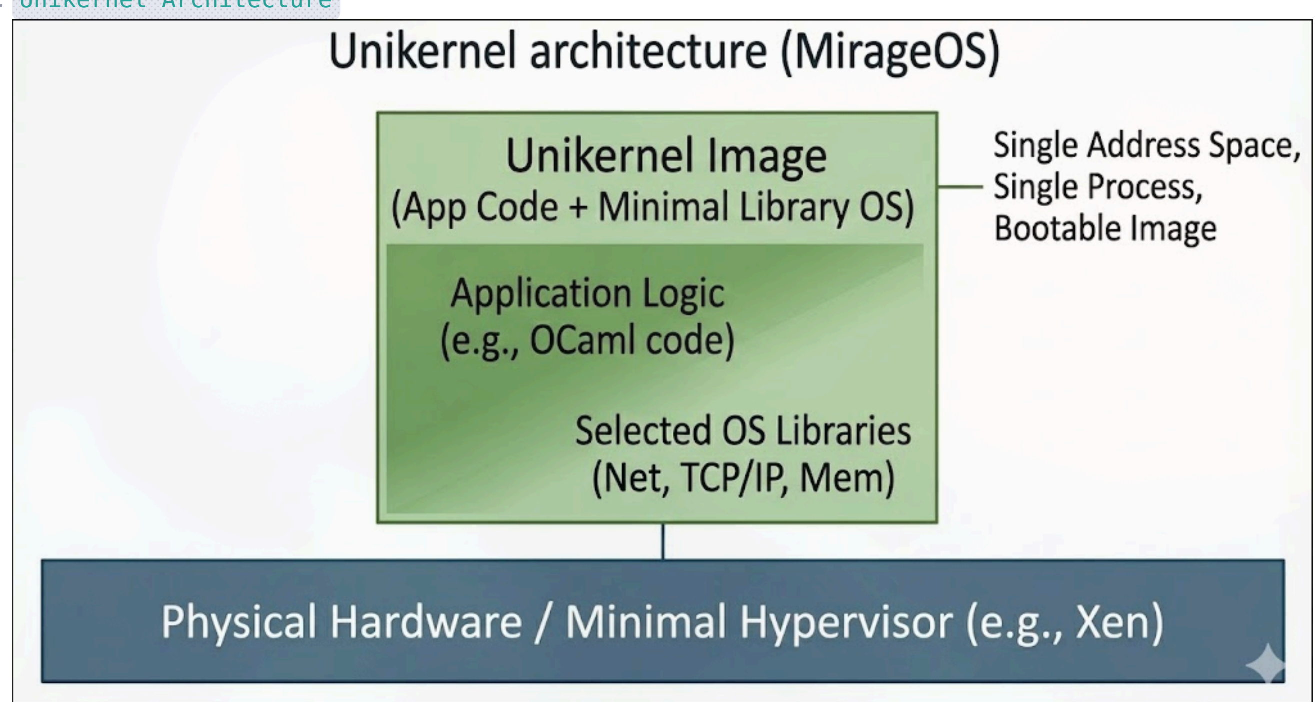
1. `bare-mental`




2. `Virtualization & Containers`



3. `Unikernel Architecture`




不知道是个人的视野狭窄,还是现状就是如此,大环境下多的不是开发就还是开发,不是前端开发,就是后端开发,或者全栈开发. 感觉真的很少听到关于操作系统方面的,就更不用说系统的形式化验证了,我感觉 `OCaml` 这种 `Functional language` 真的在欧洲那边真的比较流行,用于系统的形式化验证和编写. 上面的 `MirageOS` 就是主要是**The University of Cambridge**研发的一个 `Unikernel` 系统,主要是用 `OCaml` 编写的.

但无论计算范式如何迁移，从宏内核到微内核，再到 Unikernel，真正需要被长期维护的，始终是对系统本质的理解——调度、隔离、抽象与约束. 因此，系统性的学习不会因为技术形态的更替而终止，反而愈发必要,我也会在这条路尽可能的走下去,不管是兴趣使然或是自娱自乐。