P1: Network device driver

Ad Nunes Ribeiro (11200620), Eduardo Dias Defreyn (11203780),
Ricardo do Nascimento Boing (14200760)
Bacharelado em Ciência da Computação -- Universidade Federal de Santa Catarina (UFSC)
Campus Reitor João David Ferreira Lima, 88.040-900 -- Florianópolis -- SC -- Brasil

adnunesribeiro@gmail.com; eduardo_dududex@hotmail.com; ricardoboing.ufsc@gmail.com

1. Network device driver (i82557a)

O grupo não conseguiu implementar o driver para placa i82557a.

2. Testes no PCNet32

Considerando que não foi desenvolvido um driver, para placa i82557a, o grupo utilizou a PCNet32 para realizar os testes solicitados pelo professor. Os testes se encontram na pasta tests_sos, pois em nossas máquinas só foi possível obter sucesso na compilação (com make, make veryclean e make run APPLICATION) se houvesse apenas um teste na pasta app. Ou seja, toda vez que for executar um teste deve-se move-lo para pasta app e remover o teste existe.

2.1. Teste demonstrando a operação do driver com DMA, tanto para recepção quanto para transmissão (1 pt)

O teste implementado em dma.cc serve para validar este item. A ideia é que, em casos de não utilização do DMA, um dispositivo I/O utiliza a CPU 100% do tempo, e com DMA é possível executar outras tarefas enquanto o dispositivo I/O não retorna. Desse modo, considerando a sugestão do professor de implementar uma função fibbonacci, foi obtido o tempo da execução do fibbonacci, depois o tempo do envio da mensagem pela rede, e depois executado os dois em conjunto (através de duas threads). O resultado, mostrado na Fig. 1, demonstra o paralelismo das operações fibbonacci e I/O.

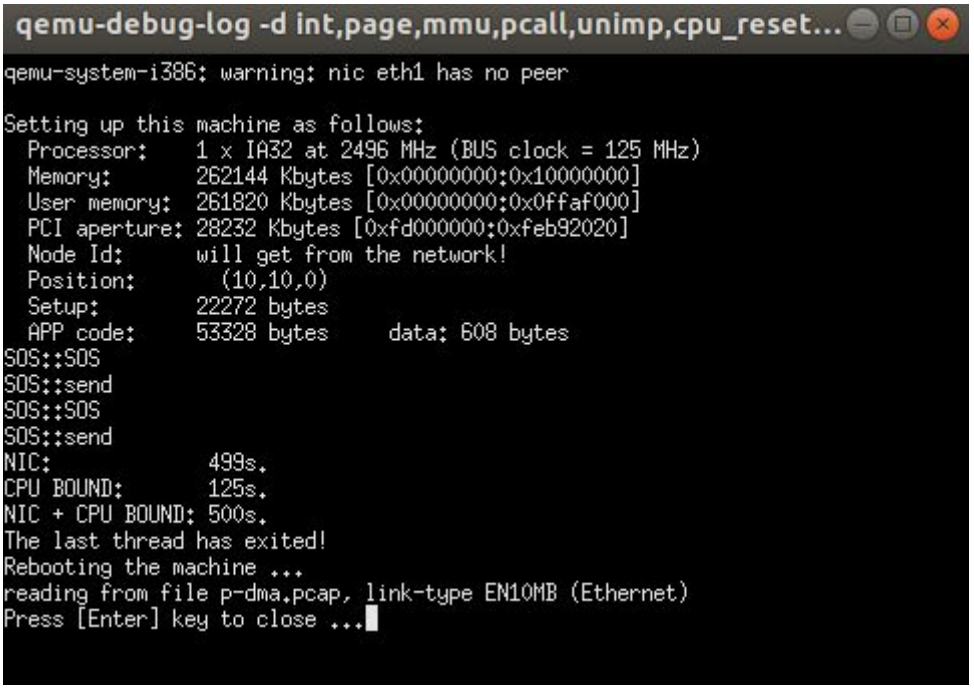

Figura 1. Teste de DMA.

2.2. Teste demonstrando o tratamento de overflow dos buffers, tanto de recepção quanto de transmissão (1 pt)

O teste overflow.cc mostra o envio de 250 pacotes de 1000 bytes de um QEMU para outro. O QEMU receptor possui um delay, antes de começar a receber pacotes, para permitir o overflow no buffer. A Fig. 2 mostra as estatísticas do QEMU receptor, onde é possível visualizar o recebimento dos 250 pacotes.

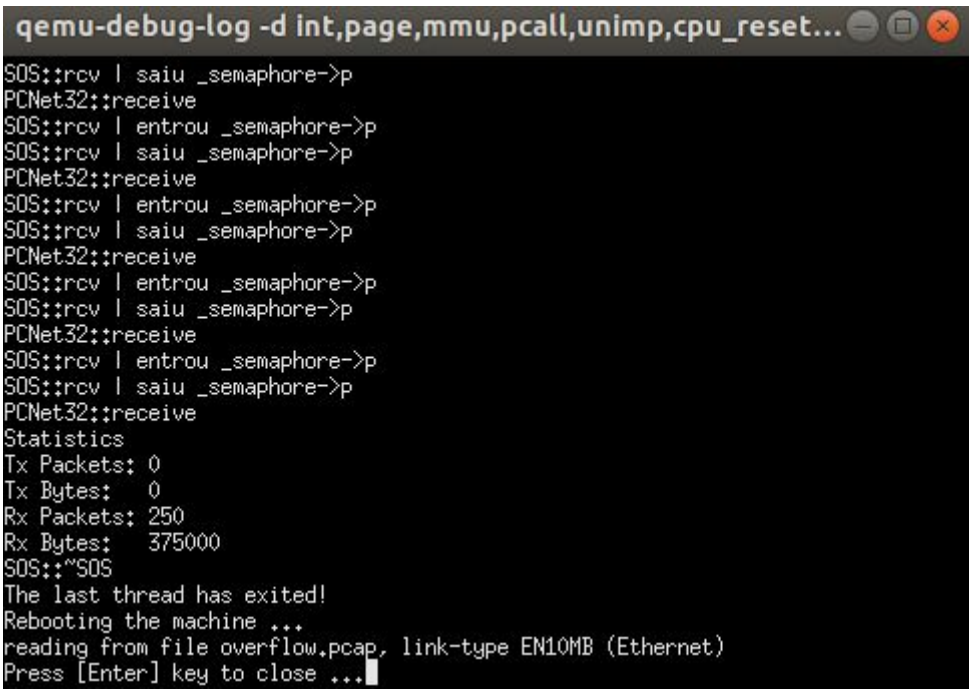

Figura 2. Resultado do teste de overflow.

2.3. Teste demonstrando o fluxo de pacotes de rede entre o driver e a aplicação, considerando o desacoplamento do mecanismo de tratamento de interrupções, o escalonamento e transferência dos dados para o escopo da aplicação (5 pt)

A classe SOS foi criada para interagir com a aplicação, com o objetivo de enviar e receber dados da rede. A SOS possui funções para enviar e receber dados, sem que haja a necessidade de utilizar diretamente o device driver. Por não conseguirmos criar a classe em um arquivo separado, utilizamos os arquivos .h e .cc de um protocolo já existente: IP (ip.h e ip.cc). Apesar de serem declarados nos mesmos arquivos, as classes SOS e IP não possuem nenhum vínculo. As Fig. 3 e Fig. 4 mostram a declaração e implementação da classe SOS.

```
415 class SOS: private NIC<Ethernet>::Observer {
416 public:
417     typedef unsigned char Protocol;
418     typedef Ethernet::Buffer Buffer;
419     typedef Data_Observer<Buffer, Protocol> Observer;
420     typedef Data_Observed<Buffer, Protocol> Observed;
421     typedef NIC<Ethernet>::Address NIC_Address;
422
423     enum {
424         PROTOCOL_SOS = 0x88,
425         PROTOCOL_SOS_1 = 0x89
426     };
427
428     SOS(unsigned short prot);
429     ~SOS();
430
431     void send(char data[]);
432     void rcv(char data[]);
433     void statistics();
434
435     static void attach(Observer * obs, const Protocol & prot) { _observed.attach(obs, prot); }
436     static void detach(Observer * obs, const Protocol & prot) { _observed.detach(obs, prot); }
437
438     static NIC_Address nic_address() { return SOS::nic->address(); }
439
440 protected:
441     void update(Ethernet::Observed * obs, const Ethernet::Protocol & prot, Buffer * buf);
442
443 protected:
444     static Observed _observed;
445     static NIC<Ethernet> * nic;
446     Semaphore* _semaphore;
447     unsigned short protocol;
448
449 };
```

Figura 3. Declaração e implementação da classe SOS.

```
197  SOS::SOS(unsigned short prot) {
198      using namespace EPOS;
199      OStream cout;
200      cout << "SOS::SOS" << endl;
201
202      protocol = prot;
203
204      SOS::nic = Traits<Ethernet>::DEVICES::Get<0>::Result::get(0);
205      SOS::nic->attach(this, protocol);
206
207      _semaphore = new Semaphore(0);
208  }
209  SOS::~SOS() {
210      using namespace EPOS;
211      OStream cout;
212      cout << "SOS::~SOS" << endl;
213
214      SOS::nic->detach(this, protocol);
215      delete _semaphore;
216  }
217  void SOS::update(NIC<Ethernet>::Observed * obs, const NIC<Ethernet>::Protocol & prot, Buffer * buf)
218  {
219      using namespace EPOS;
220      OStream cout;
221      cout << "SOS::update | _semaphore->v" << endl;
222
223      _semaphore->v();
224  }
225
226  void SOS::send(char data[]) {
227      using namespace EPOS;
228      OStream cout;
229      cout << "SOS::send" << endl;
230
231      SOS::nic->send(nic->broadcast(), protocol, data, nic->mtu());
232  }
233  void SOS::rcv(char data[]) {
234      using namespace EPOS;
235      OStream cout;
236      cout << "SOS::rcv | entrou _semaphore->p" << endl;
237
238      _semaphore->p();
239
240      cout << "SOS::rcv | saiu _semaphore->p" << endl;
241
242      NIC<Ethernet>::Address src;
243      NIC<Ethernet>::Protocol prot;
244      SOS::nic->receive(&src, &prot, data, 1000);
245  }
```

Figura 4. Implementação dos métodos da classe SOS.

Os testes implementados em conexao.cc, protocolo.cc, dma.cc e overflow.cc, na pasta tests_sos (raiz do projeto), servem para validar este item.

2.4. Teste demonstrando a operação do driver suportando a comunicação entre dois QEMUs (1 pt)

Os testes implementados em conexao.cc, protocolo.cc, dma.cc e overflow.cc demonstram a troca de mensagens entre os dois QEMUS.

2.5. Teste demonstrando a operação do driver com interrupções (1 pt)

Os testes implementados em conexao.cc, protocolo.cc, dma.cc e overflow.cc demonstram a operação do PCNet32 com interrupções. Isso porque, na classe SOS::rcv é feito o bloqueio da thread da aplicação (dando um semaphoro->p()) até que uma interrupção seja recebida e um notify chame a função update. A função update é responsável por desbloquear a thread da aplicação (dando um semaphoro->v()).

2.6. Teste demonstrando a interoperabilidade com outros protocolos (1 pt)

O teste conexao.cc utiliza um tipo de protocolo diferente daquele utilizado pelo teste protocolo.cc, para enviar/receber uma mensagem. Os dois protocolos demonstram que o PCNet32 pode ser utilizado por mais de um tipo de protocolo.