Universidade Federal de Santa Catarina – UFSC
INE5424 - Sistemas Operacionais II
Prof. Antônio Augusto Fröhlich

Seminário H:
I/O Bus Operation

Guilherme Antônio Ferreira da Silva
João Fellipe Uller

Florianópolis, 14 de Outubro de 2020

RISC-V I/O

Para lidar com dispositivos de entrada/saída, a ISA do RISC-V prevê um esquema de *Memory Mapped I/O* (MMIO), ou seja, o RISC-V se utiliza do mesmo espaço de endereçamento para endereçar tanto memória quanto dispositivos de entrada e saída. Dessa forma, a especificação da ISA não prevê nenhum tipo de instrução especial para controle ou manipulação do barramento para realizar I/O, mas sim as mesmas instruções de utilizadas para acessar a memória são aplicadas da mesma maneira para acessar endereços dentro do espaço de endereçamento que se referem a dispositivos externos.

Esse princípio se apoia na ideia de se ter uma ISA que seja simples e que possua instruções simples, uma vez que não se tem instruções adicionais para se lidar com operações de entrada e saída, apenas aquelas previamente existentes para acesso à memória. Além disso, o esquema de entrada e saída mapeado em memória também incorre num design de CPU mais simples e mais eficiente, além do fato de que utilizar as instruções regulares de memória para I/O permite utilizar qualquer um dos modos de endereçamento suportados pela CPU para endereçar esses dispositivos. Além de simplificar a programação, isso também força uma proteção nos acessos de dados através do espaço de endereçamento, evitando que *threads* do espaço do usuário acessem endereços mapeados para entrada e saída diretamente.

Memory Mapped I/O (MMIO)

A ideia do esquema de MMIO consiste no fato de que cada um dos dispositivos de entrada e saída sejam mapeados e associados para um ou mais endereços dentro do espaço de endereçamento do sistema. A partir dai, cada um dos dispositivos de I/O passa a monitorar o barramento de endereços da CPU, afim de interceptar requisições para qualquer um de seus endereços associados. É importante notar que aqui não se fala de reservar um espaço dentro da memória para comunicação com o dispositivo, mas sim de mapear os registradores de *hardware* dos dispositivos para dentro do espaço de endereçamento do processo.

Esse mapeamento do espaço de endereçamento fica a cargo do ambiente de execução, e geralmente é realizado durante o boot do sistema, sendo feito por algum *firmware* ou por alguma *Memory Management Unit* (MMU). No RISC-V, as regiões do espaço de endereçamento podem ser mapeadas como: (i) memória, (ii) I/O ou (iii) vazias, que basicamente são regiões de I/O sem permissões de acesso. Cada região então possui *Physical Memory Attributes* (PMAs) específicos que definem as operações passíveis de serem realizadas sobre aquele intervalo de endereços, sua acessibilidade, além de uma série de outras características que tratam da maneira como aquela região é tratada, como por exemplo se ela é uma região que é passível de ser otimizada via *cache* ou não.

A respeito das regiões de I/O, elas tipicamente são acessadas através de *uncached loads/stores*, isto é, é feito um *bypass* na memória *cache* e os dados são buscados diretamente dos controladores de dispositivos. Esse fato dos periféricos não poderem ser acessados da mesma maneira que a memória principal, passível de ser armazenada na *cache*, se encontra nos requisitos de consistência existentes no caso de operações de I/O, além dos possíveis efeitos colaterais que são passíveis de acontecer em leituras e escritas de dispositivos de entrada e saída. Além disso, regiões de I/O ainda podem especificar quais combinações de permissões de acesso são válidas dentro do seu intervalo, além das larguras de dados suportadas pelos dispositivos aos quais elas mapeiam.

No porte do RISC-V no emulador QEMU, o protocolo utilizado para o mapeamento de I/O é o VirtIO, que emula dispositivos externos virtuais e visa estabelecer uma API padrão para comunicação com esses dispositivos. Voltando ao porte do RISC-V, o QEMU faz o mapeamento desses dispositivos do VirtIO a partir do endereço `0x10001000` até `0x10008000`, de trás para frente, isto é, se tivermos apenas um dispositivo, ele deve estar mapeado para o endereço `0x10008000`.

Command Line

```
$info mtree
...
memory-region: system
    0000000000000000-ffffffffffffffff (prio 0, i/o): system
        ...
        0000000010001000-00000000100011ff (prio 0, i/o): virtio-mmio
        0000000010002000-00000000100021ff (prio 0, i/o): virtio-mmio
        0000000010003000-00000000100031ff (prio 0, i/o): virtio-mmio
        0000000010004000-00000000100041ff (prio 0, i/o): virtio-mmio
        0000000010005000-00000000100051ff (prio 0, i/o): virtio-mmio
        0000000010006000-00000000100061ff (prio 0, i/o): virtio-mmio
        0000000010007000-00000000100071ff (prio 0, i/o): virtio-mmio
        0000000010008000-00000000100081ff (prio 0, i/o): virtio-mmio
        ...
```

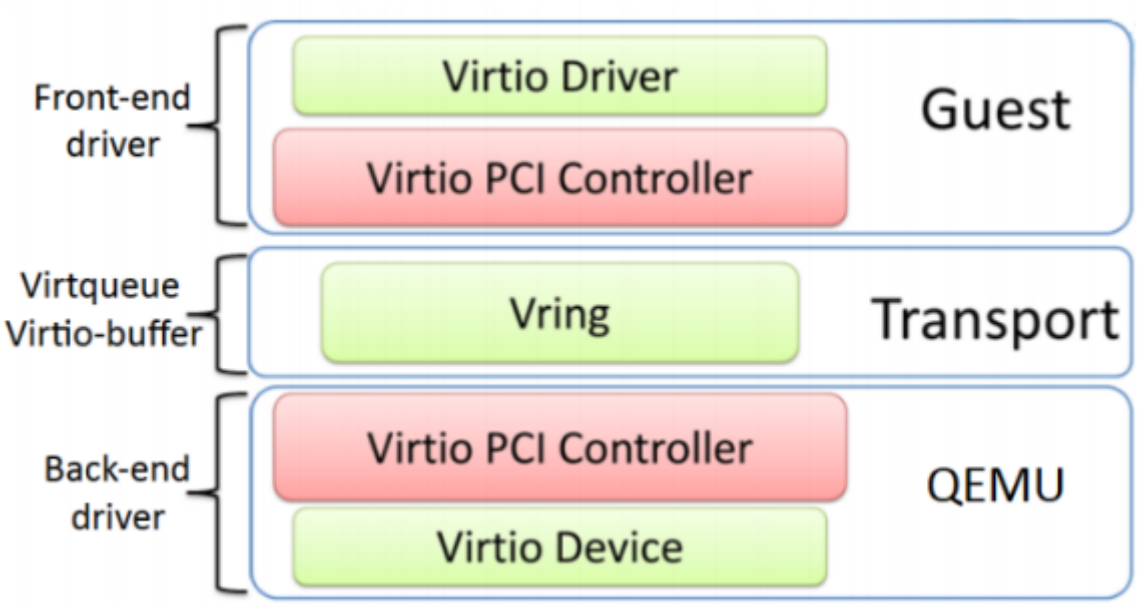
I/O Ordering

VirtIO

O VirtIO é um protocolo de comunicação entre dispositivos virtuais, criado com o objetivo de definir uma API para unificar drivers e facilitar o uso e configuração desses dispositivos, visto que existem diversos sistemas de virtualização diferentes e cada um desses realiza a sua própria implementação. Para alcançar tais objetivos, o virtIO define uma série de abstrações e estruturas, mas de maneira geral seus principais componentes de funcionamento são 3 (três): *device status field*, *feature bits* e uma interface chamada *Virtqueue*.

O *device status field* é uma sequência de bits utilizada pelo dispositivo em conjunto com o sistema operacional para realizar a inicialização do mesmo. Os bits que podem ser setados nesse campo são: **ACKNOWLEDGE**, indicando que o dispositivo foi reconhecido, **DRIVER** para indicar que a inicialização começou, **DRIVER_OK** e **FEATURES_OK** para sinalizar que a comunicação está pronta para começar. Por fim, **DEVICE_NEEDS_RESET** para indicar uma falha fatal do lado do dispositivo e **FAILED** pelo lado do sistema operacional. Já o *feature bits field* é utilizado para comunicar sobre quais features são suportadas pelo dispositivo e entrar em acordo com o sistema operacional sobre quais dessas serão utilizadas. Por exemplo, um dispositivo de interface de rede poderia escolher entre checksumming ou scatter-gather realizar o offload.

Por fim, temos a Virtqueue, uma interface alocada na memória do guest (dispositivo) e que realiza a comunicação entre dois grupos: front-end, onde os pedidos de I/O do processo usuário são realizados e então transferidos para o back-end, que os recebe e então executa as operações utilizando os dispositivos físicos.



[b]

Estrutura da Virtqueue

A Virtqueue é implementada usando uma estrutura chamada *Vring*, que consiste basicamente de 3 (três) partes: um array de descriptors, onde são inseridos os dados, um *available ring*, usado para a comunicação do sistema operacional com o dispositivo e um *used ring*, utilizado para a comunicação no sentido do dispositivo para o sistema operacional.

Os descriptors contém um endereço físico de 64 bits de buffer junto com o seu tamanho, além de um campo para flags, que podem ser duas: uma para indicar se existe um encadeamento de buffers, permitindo o uso de memória não contígua. A outra flag indica o tipo do buffer, se é *read-only* ou *write-only*. Por fim, o campo next aponta para o próximo descriptor, se o mesmo existir. Os outros dois componentes da Virtqueue são chamados de available ring e used ring. O motivo de se utilizar essa estrutura em anéis é permitir assincronicidade, fazendo com que diversas requisições possam ser feitas sem ter que esperar por uma específica finalizar primeiro.

```
struct vring_desc
{
  __u64 addr;
  __u32 len;
  __u16 flags;
  __u16 next;
};
```

Quando queremos realizar uma ação com o dispositivo, devemos preencher os valores desse descriptor e então adicionar o seu índice dentro de uma lista no available ring. Dessa forma, ao notificar o sistema operacional, o mesmo irá verificar qual descriptor deve ser lido. A estrutura de um available ring possui seu índice e uma flag, onde é possível desabilitar interrupções, enquanto a constante NUM é negociada entre o dispositivo e o sistema operacional quando o mesmo é inicializado.

```
struct vring_avail
{
  __u16 ring[NUM];
  __u16 flags;
  __u16 idx;
};
```

Ainda, temos os used rings, onde o dispositivo pode enviar algumas informações ao sistema operacional, como por exemplo indicar que uma operação foi finalizada. Sua estrutura é muito semelhante a do available ring, com a diferença que aqui o SO tem que procurar qual descriptor realizou a notificação.

```
struct vring_sed
{
  __u16 flags;
  __u16 idx;
  __u16 avail_event;
  Elem ring[NUM];
};

struct Elem {
  u32 id;
  u32 len;
}
```

Outra diferença entre o available e o used ring é que aqui, no used ring, o campo *idx* indica o primeiro elemento não usado do anel. Isso é utilizado pois dentro da estrutura 'Elem', também temos o índice do descriptor. Assim, a cada leitura realizada, o dispositivo incrementa o índice em used ring. Dessa forma, se os índices não são iguais, significa que existe ainda dados a serem lidos. Tanto o sistema operacional quanto o dispositivo acompanham o estado desses anéis, garantindo que estejam falando sobre a mesma requisição.

Por fim, para indicar onde procurar na memória pelos descriptors, os dispositivos do virtIO tem um registrador chamado *QueuePFN (Page Field Number)* , que contém o endereço de memória física para o qual foi mapeado. Com esse componentes, conseguimos então realizar todo o ciclo de uma requisição. Iniciamos apontando para o endereço de memória existente em QueuePFN e então, quando um descriptor for preenchido com dados, notificamos que uma requisição foi realizada através de um outro registrador, chamado *QueueNotify*, escrevendo um valor qualquer no mesmo. O processo se inicia e quando o mesmo finalizar, uma interrupção externa será enviada pelo dispositivo, de forma que o sistema operacional precisa apenas verificar o used ring e obter os dados.

Exemplo de configuração e uso de um dispositivo

Para iniciar o processo de uso e configuração de um dispositivo, precisamos começar lendo todo o barramento e procurar pelo chamado *Magic Value*, i.e, a string 'virt'. Neste exemplo, usando o QEMU, precisamos verificar todos os endereços entre 0x1000_1000 e 0x1000_8000, pois é onde os dispositivos virtios são alocados pelo QEMU. Encontrado esse valor, podemos ler então o valor do registrador *DeviceID*, que identifica o tipo de dispositivo. Nesse caso, procuramos pelo valor 2, para *block devices*. O trecho de código abaixo mostra um pedaço dessa leitura de endereços até encontrarmos o dispositivo desejado.

```
for addr in (MMIO_VIRTIO_START..=MMIO_VIRTIO_END).step_by(MMIO_VIRTIO_STRIDE) {
  let magicvalue;
  let device_id;
  let ptr = addr as *mut u32;

  unsafe {
    magicvalue = ptr.read_volatile();
    deviceid = ptr.add(2).read_volatile();
  }

  if MMIO_VIRTIO_MAGIC != magicvalue {
    continue;
  }
  else if 0 == deviceid {
    println!("not␣connected.");
  }
  else {
    match deviceid {
      2 => {
        if false == setup_block_device(ptr) {
          println!("setup␣failed.");
```

```
        }
        else {
          let idx = (addr - MMIO_VIRTIO_START) >> 12;
          unsafe {
            VIRTIO_DEVICES[idx] =
              Some(VirtioDevice::new_with(DeviceTypes::Block));
          }

          println!("setup␣succeeded!");
        }
      },
      _ => println!("unknown␣device␣type."),
    }
  }
}
```

Feito isso, começamos a configurar o dispositivo e sua comunicação com o sistema operacional. Iniciamos essa configuração setando o bit de **ACKNOWLEDGE** no device status, seguido do bit de **DRIVER**. Além disso, precisamos entrar em acordo com o dispositivo sobre quais features serão habilitadas, através da leitura do registrador *guest_features* e então setando o bit de **FEATURES_OK**. Por fim, setamos o bit de **DRIVER_OK** e agora o dispositivo está conectado.

O VirtIO define uma série de protocolos para cada tipo de dispositivo quando vamos fazer uma requisição, então precisamos seguir o definido para dispositivos do tipo block. Sendo assim, usaremos 3 (três) descriptors nesse processo: uma para o header, buffer e status. O do header é responsável por dizer para o dispositivo se o sistema operacional quer escrever ou ler e o endereço onde essa ação deve ocorrer. O buffer, se for de leitura, irá escrever seu valor na memória, caso seja escrita, o dispositivo irá ler o endereço na memória. Por fim, o status field guarda o resultado da requisição, que pode ter 3 (três) valores: 0 - success, 1 - failure ou 2 - unsupported operation.

Para realizar a requisição, pegamos os três descriptors que precisamos, inserimos os dados de header, buffer e status e então escrevemos o valor 0 dentro do registrador queue_notify para avisar ao dispositivo que ele deve iniciar a trabalhar na requisição. Ao finalizar a execução, o dispositivo então emite uma interrupção e recebemos de volta um used ring, o qual contém um id para identificar o descriptor usado. Isto se deve pois as requisições podem acabar numa ordem diferente do que foram enviadas, o que faz com que não tenhamos garantia de que é o mesmo descriptor e torna necessário que isso seja checado. Por fim, validado o descriptor e recebida a resposta, é possível liberar a memória da heap que foi alocada no ínicio da requisição e finalizar a mesma.

Referências

[1] *Virtual I/O Device (VIRTIO)*, abr 2019. Disponível em <https://docs.oasis-open.org/virtio/virtio/v1.1/cs01/virtio-v1.1-cs01.pdf>. Acessado em 10 Out. 2020.

[2] RISC-V Foundation. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*, mar 2019. Disponivel em: <https://riscv.org//wp-content/uploads/2019/06/riscv-spec.pdf>. Acessado em: 29 set. 2020.

[3] Rusty Russell. *virtio: Towards a De-Facto Standard For Virtual I/O Devices*. Disponível em <https://www.ozlabs.org/ rusty/virtio-spec/virtio-paper.pdf>. Acessado em 1 Out. 2020.