Relatório
Sistemas Operativos
2012/2013

Char Device Driver para a porta série

Rui Pedro Zenhas Graça
Pedro Manuel Nunes Sequeira

Faculdade de Engenharia
da Universidade do Porto

1 Introdução

Este trabalho consiste no projeto e implementação de dois *char device drivers* para a porta série (comunicação UART), um recorrendo a *polling* (SERP) e outro a interrupções (SERI). Em ambos os casos, são implementadas as funções de abertura e fecho dos ficheiros especiais associados aos dispositivos, assim como de inicialização e terminação dos módulos respetivos. São também implementadas as funções associadas às chamadas ao sistema de leitura (*read()*), destinada à receção de dados por parte do utilizador que a invoca, e de escrita (*write()*), destinada ao envio de dados por parte do utilizador que a invoca. É, por fim, implementada a chamada ao sistema *ioctl()*, que permite ao utilizador controlar determinados parâmetros associados à comunicação. Ambos os *device drivers* recorrem aos FIFOs de 16 *bytes* disponibilizados pelo hardware, que, como se verá, oferecem vantagens em ambos os casos.

2 SERP

Este primeiro *device driver* recorre, como foi anteriormente referido, a *polling*, estando, por isso, constantemente a sondar a disponibilidade de novos dados a ser lidos, no caso da função de leitura, ou da disponibilidade para o envio de novos dados, no caso da função de escrita.
Na função de leitura, esta sondagem é feita periodicamente, bloqueando caso não haja nada para ler, recorrendo à função *msleep_interruptible()*, que bloqueia por um tempo definido, e permite que seja interrompida, por exemplo, por opção do utilizador, caso este faça CTRL+C. Caso esta interrupção aconteça, a função de espera irá retornar um valor diferente de zero, que será testado pela função de leitura como condição para retornar ao utilizador. Esta espera é feita apenas caso não existam dados prontos a ler, e até que estes estejam disponíveis, ou até que o utilizador decida interromper. Caso a *flag* O_NONBLOCK tiver sido utilizada na abertura do ficheiro, esta espera nunca irá ocorrer, e, caso não haja caracteres a ler, será retornado -EAGAIN. Os caracteres recebidos são guardados num *buffer* ao nível do *kernel*, alocado para o número de caracteres pedidos pelo utilizador. Caso este número seja excessivamente grande, será impossível alocar memória suficiente, e será retornado -ENOMEM. Será, depois, efetuada a sondagem por novos dados (retornando caso estes não existam e a *flag* O_NONBLOCK estiver ativa), e, após a receção, é feita a transferência para o *buffer* do utilizador. Esta transferência é feita *byte* a *byte*. A partir do momento que um *byte* é recebido, a função apenas retorna no fim de um *burst* de carateres recebidos, bloqueando (pela função *msleep_interruptible()*) por um número limitado de vezes se não houver dados recebidos depois de o ultimo *byte* ter sido enviado ao utilizador, e dado por terminado o *burst*, retornando ao utilizador, caso no fim deste número limitado de bloqueios não houver novos dados. O teste de novos dados é feito testando a *flag* UART_LSR_DR. É ainda feito o teste a erros de *overrun*, verificando a *flag* UART_LSR_OE. A utilização de FIFOs do *hardware* permite que a probabilidade de *overrun* seja drasticamente reduzida, e permite, até, reduzir o *busy waiting*, uma vez que o tempo de bloqueio pode ser aumentado em relação ao que tinha anteriormente sido usado sem a utilização de FIFOs.
A função de escrita tira vantagem modo de operação com FIFOs, uma vez que é possível enviar os dados em grupos de 16 de cada vez que a *flag* de de FIFO vazio é ativada. Caso a *flag* não esteja ativada, deve ser efetuada uma espera, que, neste caso, recorre à função *schedule()*. Em caso de a *flag* O_NONBLOCK estar ativada, esta espera nunca é feita e a função retorna -EAGAIN em caso de nada ter sido escrito, ou o número de caracteres escritos, em caso de escrita parcial. Neste caso, dadas as características do FIFO, e da implementação da função, o *buffer* ao nível do

kernel de interface entre o *buffer* do utilizador e a saída foi implementado com apenas 16 *bytes* (tamanho do FIFO), o que permite a escrita de grandes volumes de dados sem que haja problemas de memória. Neste caso, devido a esta implementação, a transferência de dados do utilizador para o *buffer* ocorre ciclicamente, sendo que dentro de cada um destes ciclos, ocorre um novo ciclo que envia para o buffer de saída os dados recebidos do utilizador. Para este funcionamento, é necessário dividir os dados enviados pelo utilizador em blocos de 16 *bytes*, sendo necessário guardar o tamanho do último bloco, que num caso geral será inferior a 16, sendo que por isso, não se inclui este caso dentro do ciclo para os restantes blocos, mas sim após este.
Em ambas as funções recorre-se à exclusão mútua (recorrendo a um semáforo para cada função, para cada dispositivo aberto). Esta exclusão mútua é essencial para assegurar um fluxo de dados correto, visto que, caso não ocorresse, dando-se o facto de mais do que um processo estar a escrever ao mesmo tempo, a informação recebida no outro lado da comunicação iria receber intercaladamente dados enviados por um processo ou por outro, sem uma sequência definida, visto que esta depende do escalonamento. Uma situação semelhante iria ser verificada numa leitura por vários processos ao mesmo tempo. A exclusão mútua garante, assim, um fluxo de dados correto. Cada chamada *write()* será lida garantidamente sem que seja intercalada com uma outra, e cada chamada *read()* irá, garantidamente, ler dados sequenciais. O envio em *full-duplex* está também assegurado, visto que o funcionamento de cada uma das funções *read()* e *write()* é totalmente independente da outra, visto que nenhum recurso é partilhado entre elas, dado que os *buffers* e os semáforos utilizados são diferentes.
A função *open()* encarrega-se de guardar o endereço da *struct* do dispositivo associado ao ficheiro aberto, e de assegurar que apenas um utilizador pode ter um dispositivo aberto ao mesmo tempo. Esta solução baseia-se na proposta pelo livro *Linux Device Drivers* (Corbet, Rubini, Kroah-Hartman (2005)). A restrição a um único utilizador é uma melhor escolha do que a restrição a um único processo, visto que o utilizador pode optar por ter, por exemplo, um processo a abrir ler dados e outro a escrever. Além do utilizador que detém a possibilidade de acesso ao dispositivo num dado momento, é dada a possibilidade ao administrador de abrir um dispositivo, independentemente de quem quer que esteja com ele aberto. O registo do número de processos a utilizar o dispositivo é armazenado numa variável, que é incrementada no *open()* e decrementada no *release()*. Ao contrário da solução apresentada no livro, a solução aqui proposta não variáveis globais, mas sim variáveis associadas ao dispositivo, pertencentes à sua *struct*, o que permite a escalabilidade do código, apresentando uma solução mais geral. A função *release()* apenas se encarrega de decrementar o número de processos a utilizar o dispositivo.
A função de inicialização do módulo encarrega-se de alocar os dispositivos requeridos (neste caso é apenas um). Esta alocação é feita dinamicamente. É também alocado dinamicamente o vetor que contém as estruturas associadas a todos os dispositivos utilizados (mais uma vez, neste caso é apenas um, mas a solução apresentada é escalável), sendo realizado um ciclo para inicializar os seus elementos. Estes elementos consistem numa estrutura *cdev*, à qual é atribuída a estrutura *file_ operations* onde constam as funções utilizadas, os semáforos já referidos para leitura e escrita, as variáveis necessárias ao controlo de acesso a um único utilizador (registo de qual é o utilizador, contador do número de processos que estão a utilizar o dispositivo, e *lock*), e o *minor* associado ao dispositivo. No fim de cada iteração do ciclo de inicialização, cada dispositivo é registado.
Por último, a função de inicialização encarrega-se de fazer *request* ao dispositivo e configurar a comunicação para as opções *default*, desabilitando interrupções e habilitando os FIFOs.
A função de terminação do módulo encarrega-se de desfazer tudo aquilo que foi feito pela de inicialização: libertar todos os dispositivos, a memória alocada, e terminar o registo de todos os

dispositivos.
A função *ioctl()* permite ao utilizador alterar as configurações da comunicação, assim como ver quais são as configurações que estão a ser usadas. Os comandos implementados são SERP_WLEN, para o qual o utilizador deve enviar um valor entre 5 e 8 que irá corresponder ao número de *bits* por palavra, SERP_NUM_SB, que deve espera um valor de 1 ou 2, que será o número de *bits* de terminação por palavra, SERP_PARITY, que define a paridade de acordo com as *flags* NONEPAR, ODDPAR, EVENPAR, ONEPAR ou ZEROPAR; SERP_BAUD, que recebe um valor entre 1 e 2ź-1, que será configurado como o divisor da *baudrate*, SERP_SBC, que espera SET ou UNSET e ativa ou desativa o *break control*. Por fim, os comandos SERP_GETBAUD e SERP_GETLCR permitem obter a *baudrate* atual e o estado do registo de controlo, respetivamente.
Mais do que em qualquer circunstância, uma vez que se trabalha aqui ao nível do *kernel*, é bastante importante verificar todo os valores retornados, procedendo da melhor forma caso ocorra algum erro. É crucial verificar o valor de retorno das funções invocadas, e retornar constantes de erro adequadas caso estas falhem. É também importante que, em caso de erro, ou quando deixa de ser necessária, toda a memória alocada seja libertada. Da mesma forma, se ocorrer algum erro que impeça a correta instalação do módulo, todos os recursos reservados e todos os registos feitos devem ser anulados. Isto foi tido em especial consideração, e procurou-se sempre que, em caso de erro, a saída anulasse tudo o que deveria seria ser anulado, e libertasse tudo o que deveria ser libertado.
As variáveis globais foram reduzidas a um apontador para *struct*, que contém todas as variáveis necessárias ao funcionamento de cada um dos dispositivos, a uma *struct file_operations*, que indica quais as funções que devem ser associadas às chamadas ao sistema, e a um inteiro que guarda o número do major a ser utilizado. Em vez deste inteiro, poderia ser usada uma *struct dev_t*.
As funções simples de escrita e leitura foram testadas recorrendo simplesmente a ficheiros que realizassem várias chamadas ao sistema de leituras e escritas de vários caracteres, sendo que para o teste da *flag* O_NONBLOCK apenas foi introduzida esta *flag* na abertura do ficheiro. Alem de ter sido testados recorrendo ao ficheiro *pipe* de comunicação entre o *host* e a máquina virtual e ao programa *usocat*, o *device driver* foi testado com uma comunicação série física, obtida por curto-circuito da saída da porta série à sua entrada. A restrição a um único utilizador foi testada com recurso a um programa que faz *fork()* e tenta aceder a um dispositivo em utilização a partir de um outro utilizador, ou então a partir do mesmo utilizador que já está a utilizar o recurso. O teste à funcionalidade de *full-duplex* foi feita com a porta série física em curto-circuito, com dois processos do mesmo utilizador a aceder ao dispositivo em simultâneo, um a ler em ciclo infinito, e o outro a escrever strings de dimensões elevadas. Para verificar a existência um fluxo de dados correto, utilizou-se não um, mas dois processos de escrita, cada um com a sua própria *string*. Visto que nada restringe o acesso em simultâneo às funções de *read()* e *write()*, e que os dados foram recebidos corretamente para comunicações de cerca de 32000 caracteres, a conclusão a tirar é um correto funcionamento deste modo de operação. A chamada *ioctl()* foi testada enviando vários comandos com alterações e verificando a resposta obtida aos comandos de obtenção das configurações atuais. Todas as restantes funcionalidades foram testadas com programas simples de leitura (geralmente em ciclo infinito), ou de escrita de *strings*.

3 SERI

Este *device driver* tem por base o SERP, descrito anteriormente. É, contudo, consideravelmente diferente deste, dado que deixa de existir uma sondagem constante à disponibilidade de recursos ou à existência de dados, para que exista um controlo baseado em interrupções.

Para a transferência de dados, são usadas as *flags* de configuração das interrupções UART_IER_RDI e UART_IER_THRI. Sempre que existe uma interrupção, o *handler* verifica o estado do registo UART_IIR, e caso a interrupção tenha sido causada pela chegada de informação, o *handler* procede à transferência dos dados recebidos para um *buffer* ao nível do *kernel*. A função de leitura limita-se, assim, a verificar se existem dados neste *buffer*, e a bloquear numa *wait queue* caso não existam, ou a passa-los para o utilizador, quando estes existem. O *buffer* utilizado ao nível do *kernel* é implementado com um *kfifo*, que, recorrendo a *spinlocks*, evita *race conditions*. A utilização de FIFOs do hardware, permite que a interrupção só surja quando exista um determinado número de *bytes* em espera, sendo que neste caso o número utilizado foi 8. Permite-se, assim, evitar a sobrecarga de interrupções. Em cada interrupção para leitura, lê-se até não haver mais nada no FIFO de entrada e guarda-se tudo o que foi lido no *kfifo* de leitura.
Para o *write()*, sugere-se duas implementações, uma delas com três variâncias. Uma delas (modo 0) é em tudo semelhante à implementada no SERP, com a diferença que o *polling* é removido, sendo que, caso não possa escrever, o processo bloqueia numa *wait queue*, sendo desbloqueado quando ocorre a interrupção que indica que a escrita é possível. A outra implementação é um processo inverso ao do *read()*. A função de escrita guarda num *buffer* ao nível do *kernel* os dados a escrever, e estes vão sendo enviados para um *kfifo* de saída à medida que existe espaço disponível. O *handler* de interrupção limita-se a colocar 16 *bytes* no FIFO de saída sempre que a interrupção é gerada. Para que a interrupção seja gerada, é necessário que, sempre que o *kfifo* esteja vazio, o primeiro *byte* a enviar seja enviado dentro da função *write()* (e após a escrita no *kfifo*), o que, quando a escrita escrita for terminada, irá disparar a interrupção. Uma vez que o *kfifo* opera em exclusão mútua, e que a interrupção nunca irá ceder o CPU antes de terminar a escrita, está garantido o correto funcionamento deste processo, garantido-se que, se na função *write()*, o número de elementos no *kfifo* é 0, existe a necessidade de enviar um *byte* para que a interrupção volte a ser disparada. Esta versão possui três variâncias, relacionadas com a espera pela terminação dos envios. Numa delas, a espera é feita dentro da função de escrita (modo 2), onde ocorrerá um bloqueio até que o *kfifo* seja esvaziado, noutra (modo 1), a espera é feita apenas quando o utilizador liberta o ficheiro, sendo que a função *write()* retorna logo após ter guardado tudo no *kfifo*, e, quando o utilizador fecha o ficheiro do dispositivo, caso o kfifo não esteja já vazio, ocorre um bloqueio até que isso aconteça. Por fim, numa ultima implementação (modo 3), este bloqueio não ocorre, e, caso o utilizador feche o ficheiro antes de o *kfifo* ter sido esvaziado, parte da informação é perdida. Caso a *flag* CHANGEMOD tenha sido ativada na compilação do módulo, o utilizador pode, através do comando SERI_WRITEMODE na função *ioctl()*, configurar a implementação desejada, sendo que apenas lhe é permitido alterar o modo de escrita quando apenas um processo tem o ficheiro aberto ao mesmo tempo. A configuração de origem do módulo aqui implementado utiliza esta *flag* inativa e usa o modo de operação 1.
Visto que dentro da função de fecho de ficheiros, no modo de operação de escrita 1, pode ocorrer um bloqueio, substituiu-se por semáforos os spinlocks que asseguravam a exclusão mutua desta região, evitando assim períodos longos de espera ativa.
Além das variáveis existentes no caso do SERP, a *struct* de cada dispositivo tem agora os dois *kfifos*, os *spinlocks* a eles associados, as variáveis de *wait queue head* associadas à leitura e à escrita, e um *buffer* para transferência ao nível do *kernel* (que evita que a os valores do *kfifo* sejam retirados *byte* a *byte*). Requer-se ainda que o registo do *interrupt handler*, que é feito na abertura do ficheiro dispositivo, quando este é aberto e o contador do número de processos que o estavam a utilizar era 0. Nesta circunstância, faz-se também a alocação de memória nos *kfifo* (alocando o de leitura caso o ficheiro tenha sido aberto para leitura, e/ou o de escrita, caso tenha sido aberto para escrita).

Em caso de escrita, aloca-se também o *buffer* temporário utilizado no *handler* das interrupções. Estas operações são, evidentemente, desfeitas na função de *release()*, ou então em caso de erro numa delas. A interrupção é registada como *shared*, o que, neste caso, não faz qualquer diferença, mas corresponde, em geral, à situação esperada num caso geral. Os testes realizados para este *device driver* foram semelhantes aos anteriores, a menos dos que recorriam à utilização da porta série física. Embora não haja motivo aparente para que o funcionamento não seja *full-duplex*, essa característica não foi testada.

4 Implementação dos vários pontos do projeto final

4.1 Minimize global variables

As variáveis globais foram minimizadas em ambos os *device drivers*

4.2 Add ioctl operations to configure communication parameters

Implementado de igual forma em ambos os *device drivers*

4.3 Allow for interrupting a process inside a read() syscall

Implementado em ambos os *device drivers*

4.4 Allow reading more than one character per read() syscall invocation.

Implementado em ambos os *device drivers*, embora a implementação com interrupções seja mais eficiente

4.5 Honor the O_NONBLOCK flag.

Totalmente implementada no SERP, mas apenas parcialmente no SERI

4.6 Add support for the select() syscall

Não implementada

4.7 Add access control to prevent more than one "user" to access the serial port

Igualmente implementada em ambos os *device drivers*

4.8 Add FIFO support

Implementado em ambos os *device drivers*, embora a vantagem seja mais clara no SERI, visto a possibilidade de interrupções para o FIFO a um valor de dados recebidos definido

4.9 Add full-duplex operation

Implementado em ambos os casos, mas apenas testado no SERP

5 Divisão de trabalho

Rui Graça - Todo o projeto, implementação, teste e depuração de todo o código utilizado. Redação do relatório
Pedro Sequeira - Realização alternativa à que é aqui proposta das funcionalidades até ao final do Lab3. Atribuição de parte das constantes de erro adequadas aos retornos no SERP (e consequentemente das que são igualmente implementadas no SERI)