3º Trabalho Prático

MC404 - Organização Básica de Computadores e Linguagens de Montagem

Prof. Diego Aranha
(Adaptado do Prof. Edson Borin)

2º Semestre de 2014

1 Introdução

O trabalho consistirá em implementar um escalonador de tarefas, isto é, a parte do sistema operacional responsável por alternar entre os processos que estão executando a cada momento na CPU do computador. Para que a implementação do escalonador fique completa, você deverá implementar as seguintes chamadas de sistema: `write()`, `fork()`, `getpid()` e `exit()`. Mais uma vez, apenas o simulador ARM será usado, e não as placas i.MX53.

2 Método

Na convenção do ARMv7 (*ABI - Application Binary Interface*), para realizar uma chamada ao sistema, você deve colocar o número da chamada de sistema no registrador R7, e os parâmetros seguem a mesma convenção de uma chamada de função comum (devem estar nos registradores R0 a R3); o valor de retorno é passado via registrador R0. Para realizar a chamada de sistema, o código de usuário utiliza a instrução `svc 0x0`. Esta instrução irá gerar uma exceção e fará com que o registrador PC aponte para a posição `base_vet + 0x08`, em que `base_vet` é a base do vetor de interrupções. Nesse ponto então, o processador troca o modo para SUPERVISOR.

No entanto, você recebe em R7 um valor que corresponde ao número da chamada que se deseja chamar. O seu tratador de chamadas de sistema deve, portanto, analisar o valor contido nesse registrador e selecionar a rotina de tratamento adequada (`write`, `exit`, `fork` ou `getpid`). Lembre-se que, para retornar do tratador de chamadas de sistema para o código do usuário que invocou a chamada, você deve utilizar a instrução especial `movs pc, lr` que além de retornar ao código do usuário, recupera o registrador CPSR original, modificando o modo do processador para USUÁRIO.

3 Especificação

3.1 Chamada de sistema `write`

A lista abaixo apresenta os parâmetros da chamada `write`:

- R0: descritor de arquivo;
- R1: Ponteiro para a posição de memória que contém os dados a serem escritos.
- R2: Quantidade de *bytes* a serem escritos.
- R7: deve assumir o valor 4.

Não é necessário nesse trabalho que seu sistema administre descritores de arquivos. Portanto, ignore o primeiro parâmetro da chamada de sistema `write`, que é passado no registrador R0. Ou seja, independente do valor do descritor de arquivo, a sua chamada deve escrever os *bytes* no dispositivo UART. Essa é uma pequena simplificação que deve ser adotada no seu sistema em relação a um sistema operacional real.

A chamada de sistema deve então escrever R2 *bytes* do *buffer* cujo ponteiro está em R1 na UART, e retornar o número de *bytes* escritos com sucesso em R0. Um valor de retorno igual a 0 indica que nenhum *byte* foi escrito, e o valor -1 indica erro. A seção seguinte apresenta detalhes sobre como enviar *bytes* para o dispositivo UART.

3.2 Porta serial

Para utilizar a interface de comunicação serial UART, é preciso primeiramente inicializar o dispositivo UART, assim como foi feito com o TZIC e GPT no laboratório anterior. Para inicializar tal dispositivo, deve-se seguir os passos de 1 a 7 da seção 75.5.1 no *datasheet* da UART, encontrado na seção do Trabalho 3 disponível no *site* da disciplina. Note que esse documento é bem grande (mais de 70 páginas), mas você apenas irá utilizar o exemplo de inicialização e os endereços de memória dos registradores da UART (seção 75.3, sétima página). Observe que o *datasheet* apresenta informações das 4 UARTs das placas i.MX53, contudo o simulador ARM apenas implementa a UART1 - assim, você sempre deve utilizar os valores referentes à UART1.

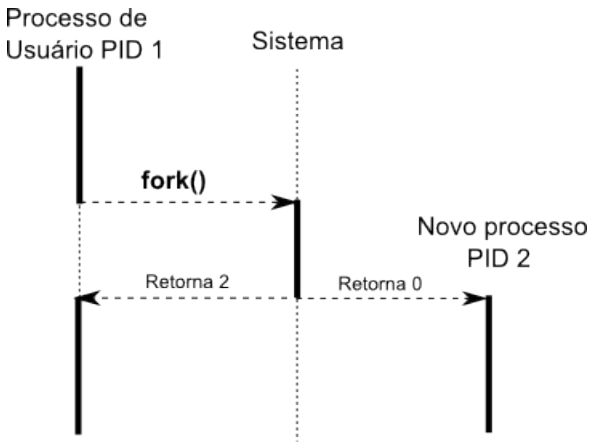
Para transmitir dados pela porta serial sem utilizar interrupções, verifique se o bit TRDY (*transmitter ready*), que é o bit 13 do registrador `UART STATUS REGISTER 1` (`UART1_USR1`), está definido como 1. Em caso positivo, temos que a ocupação da fila de transmissão (`TX_FIFO`) está abaixo de um nível de segurança pré-configurado (por você mesmo: lembre-se que você configurou a `UART` seguindo os passos do datasheet!). Ou seja, você pode escrever um *byte* para transmissão sem correr o risco de perder o dado pelo fato da fila estar cheia. Essa fila possui 32 posições e armazena os caracteres que

estão pendentes para serem enviados - quando um *byte* é transmitido, um caractere é removido da fila.

Para escrever um caractere na fila `TX_FIFO` e escaloná-lo para ser transmitido, basta realizar uma escrita no registrador `UART1_UTXD`. Ao escrever nesse registrador, o dado vai para a fila `TX_FIFO`. Caso a ocupação da fila fique alta (acima do nível de segurança), o bit `TRDY` é automaticamente desligado (passa a valer 0). Portanto, uma forma de realizar escrita pela UART é implementar um laço que escreve byte por byte, realizando sucessivas escritas em `UART1_UTXD`, sempre que o bit `TRDY` for igual a 1. A condição de parada do laço ocorre caso o número de *bytes* escritos já tenha alcançado o número solicitado pelo usuário no momento da chamada da chamada `write`. Lembre-se que o relógio (*clock*) do processador é MUITO mais alto do que o relógio dos periféricos, assim a escrita ocorre de forma "lenta"se levarmos em conta o número de instruções executadas pelo processador nesse tempo.

3.3 Gerência de processos

O seu sistema será similar a um sistema operacional verdadeiro, pois contará com a capacidade de rodar mais de um programa de usuário (ou processo) ao mesmo tempo. Este tipo de sistema operacional é chamado de sistema multi-tarefa. Em sistemas POSIX, a maneira de se criar novos processos é através da chamada de sistema `fork`:



Consulte a documentação da chamada `fork` através do comando `man 2 fork`. Observe ainda que o valor a ser passado no registrador R7 é 2.

As barras mais escuras representam o fluxo de código executado no processador. Repare que a partir da chamada para `fork`, temos 2 processos rodando ao mesmo tempo no processador. Na prática, isso não acontece, a não ser que o sistema tenha múltiplos núcleos ou múltiplos processadores. Um dos métodos usados para se implementar um sistema multi-tarefa num processador único é um escalonador preemptivo: tal escalonador alterna entre a execução de processos do usuário a cada unidade temporal denominada *time slice*; por exemplo, podemos ter uma fatia de 50 ms, ou seja, a cada 50 ms o código do usuário é interrompido e o controle é passado ao sistema operacional,

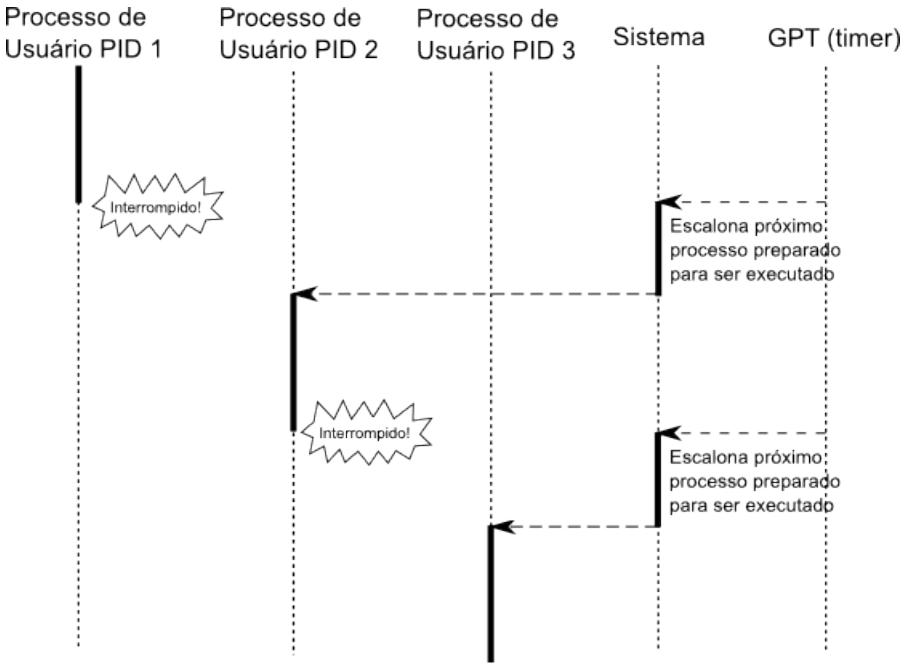
para que este possivelmente escalone outro processo de usuário. O escalonador baseado em preempção pode ser implementado em sistemas que dispõem de interrupções por tempo (exatamente o mecanismo usado no Laboratório 8).

Nesse trabalho, para todos os efeitos, o PID (process ID) é um inteiro de 32 *bits* que seu sistema deve utilizar para identificar um processo. Após o seu sistema ser inicializado (ou seja, após tratar a interrupção *reset*, inicializar dispositivos, etc), ele deverá fazer um salto para o endereço do início da seção de código do programa usuário, para que este comece a executar. O programa usuário fornecido, `DummyUSER`, chamará diversas vezes `fork` para se replicar, e nesses pontos o seu sistema deve interromper a execução do programa e fazer o tratamento de forma apropriada, como detalhado abaixo. Vamos usar aqui a seguinte convenção: o endereço de entrada de código do programa usuário será `0x77802000` (já utilizado em laboratórios anteriores).

3.4 Preempção

O comportamento esperado de um escalonador baseado em preempção é o de tomar o controle do processador forçadamente por uma interrupção de um temporizador em hardware, como o GPT (*General Purpose Timer*). Assim, para o programa usuário é como se ele fosse o único programa rodando no sistema. Após um tempo executando no processador, o temporizador provoca uma interrupção que passa o controle ao sistema operacional - o tratador de interrupções chama então o escalonador. Ao invés de voltar a execução para o processo interrompido, outro processo é colocado para ser executado no lugar. Desse modo, cada processo executa por um período de tempo predeterminado pelo temporizador (*time slice*).

A política usada nesse trabalho para escolher qual o próximo processo a ser executado é a denominada *round robin*, em que se permuta entre os processos com igual prioridade. Suponha que o seu sistema possua 3 processos ativos, isto é, 3 programas usuário diferentes - a situação mostrada nas figuras abaixo ilustra o funcionamento do escalonador usando essa política:



O seu sistema deve comportar no máximo 8 processos. Se uma chamada de sistema `fork` for realizada e o sistema já está na capacidade máxima, com 8 processos, um erro deverá ser retornado.

3.5 Troca de contexto

Lembre-se que você precisa guardar o contexto de execução de cada processo. Para isso, crie um vetor com 8 posições. Cada contexto é composto pelos registradores (inclusive o CPSR), salvos no momento em que o processo foi interrompido pelo escalonador preemptivo.

Este vetor de contextos deve guardar um *snapshot* ("fotografia") de como estavam os registradores de um processo quando ele foi interrompido pela última vez. Dessa forma, quando o processo for escolhido para ser executado em um futuro próximo, todos os seus registradores (seu contexto) estarão intactos e serão restaurados, como se o programa nunca tivesse sido interrompido. A grande vantagem do escalonador preemptivo é justamente ser transparente ao programa.

Na inicialização do seu sistema, 8 pilhas devem ser pré-alocadas em posições distintas da memória, uma para cada possível processo - cada novo processo recebe uma pilha diferente. Lembre-se de que pilha cresce para baixo. Uma sugestão para os endereços é dada abaixo.

Utilize o modelo abaixo como referência para inicializar a pilha de execução de cada programa.

```
.set SVC_STACK, 0x77701000
.set UND_STACK, 0x77702000
```

Endereço	Descrição
0x7770D000	Início da pilha do PID 1
0x7770C800	Início da pilha do modo supervisor de PID 1
0x7770C000	Início da pilha do PID 2
0x7770B800	Início da pilha do modo supervisor de PID 2
0x7770B000	Início da pilha do PID 3
0x7770A800	Início da pilha do modo supervisor de PID 3
0x7770A000	Início da pilha do PID 4
0x77709800	Início da pilha do modo supervisor de PID 4
0x77709000	Início da pilha do PID 5
0x77708800	Início da pilha do modo supervisor de PID 5
0x77708000	Início da pilha do PID 6
0x77707800	Início da pilha do modo supervisor de PID 6
0x77707000	Início da pilha do PID 7
0x77706800	Início da pilha do modo supervisor de PID 7
0x77706000	Início da pilha do PID 8
0x77705800	Início da pilha do modo supervisor de PID 8
0x77705000	Início da pilha do modo FIQ.
0x77704000	Início da pilha do modo IRQ.
0x77703000	Início da pilha do modo ABT.
0x77702000	Início da pilha do modo UND.
0x77701000	Início da pilha do modo SVC.

```
.set ABT_STACK, 0x77703000
.set IRQ_STACK, 0x77704000
.set FIQ_STACK, 0x77705000
.set USR_STACK, 0x77706000

@Configure stacks for all modes
 ldr sp, =SVC_STACK
 msr CPSR_c, #0xDF  @ Enter system mode, FIQ/IRQ disabled
 ldr sp, =USR_STACK
 msr CPSR_c, #0xD1  @ Enter FIQ mode, FIQ/IRQ disabled
 ldr sp, =FIQ_STACK
 msr CPSR_c, #0xD2  @ Enter IRQ mode, FIQ/IRQ disabled
 ldr sp, =IRQ_STACK
 msr CPSR_c, #0xD7  @ Enter abort mode, FIQ/IRQ disabled
 ldr sp, =ABT_STACK
 msr CPSR_c, #0xDB  @ Enter undefined mode, FIQ/IRQ disabled
 ldr sp, =UND_STACK
```

3.6 Recuperação de contexto

Repare que a troca de contexto pode interromper a realização de uma chamada de sistema de um processo (por exemplo, processo X). Então, a pilha de supervisor de X não pode interferir na realização de uma chamada por parte de outro processo (por exemplo, processo Y). Se todos os processos compartilhassem a mesma pilha de supervisor, uma intercalação de X para Y e em seguida para X novamente, sempre interrompendo um processo no meio de uma chamada, poderia arruinar o estado correto para a execução da syscall de X, pois Y provavelmente deixaria a pilha supervisor em um estado desconhecido para X. Assim, cada processo tem uma pilha de usuário e uma pilha de modo supervisor, como pode ser visto na tabela acima.

Para recuperar o contexto completamente, seu código deve entrar em modo SYSTEM e recuperar os registradores de usuário, inclusive a pilha de modo usuário. Em seguida, o seu algoritmo para recuperação de contexto deve entrar em modo SUPERVISOR e reescrever o registrador de pilha desse modo com a pilha supervisor particular desse processo. Por último, você deve recuperar o CPSR e o PC, que determinarão os 2 últimos fatores cruciais para uma completa recuperação do contexto: onde o código estava executando da última vez antes de ser interrompido, e o modo em que estava no CPSR (supervisor ou usuário).

Perceba por fim que no momento em que a chamada à `fork` é realizada, o novo processo criado possui exatamente o mesmo contexto do processo pai (que realizou a chamada). A única diferença é que quando o novo processo (filho) for executar, o valor de retorno da `fork` será 0, enquanto que no momento em que o processo velho (pai) for escalonado para execução, o valor de retorno da chamada de sistema será o PID do processo filho criado.

3.7 Outras chamadas

A chamada `getpid()` simplesmente retorna a identificação (PID) do processo que a chamar. Para responder a essa chamada, basta consultar seu escalonador para conhecer qual o número do processo em execução atualmente. Observe que nesse trabalho esse número só pode assumir os valores de 1 a 8. O valor a ser passado no registrador R7 é 0x14. Consulte a documentação da chamada `getpid()` através do comando `man 2 getpid`.

A última chamada a ser implementada, `exit()`, simplesmente encerra a execução do processo que a chamou e libera o seu número de PID para ser usado por um novo processo a ser eventualmente criado por `fork()`. O número da chamada `exit()` a ser passado no registrador R7 é 0x1. Consulte a documentação da chamada `exit()` através do comando `man 2 exit`.

4 Dicas

Esse trabalho não exige grande quantidade de codificação; contudo, é preciso ficar atento sobre todos os casos em que você pode ser interrompido e não conseguirá recuperar facilmente o contexto, e cuidar para que o código fique correto mesmo nesses casos. Você pode definir regiões críticas em que as interrupções são desabilitadas temporariamente - um exemplo disso deve ocorrer na chamada `write`, pois enquanto o texto de um processo está sendo escrito, não é desejável que ele seja interrompido e outro processo comece a escrever no lugar. Por isso pode ser interessante desabilitar interrupções no laço de escrita da UART.

Outro ponto importante no trabalho é o conceito de reentrância. Você deve prestar atenção para o uso de variáveis globais no código de sistema, pois o seu código deve funcionar mesmo quando for interrompido e outro processo chamar a mesma função (entrar novamente na função). Repare que se você utiliza variáveis globais sem proteger o acesso com travas, você pode estar no meio da atualização de uma estrutura de dados complexa e ser interrompido - em seguida outro processo chama a mesma função, que irá encontrar essa estrutura global em um estado inconsistente e então poderá falhar.

Finalmente, note que não é necessário implementar o escalonador usando a estrutura de dados denominada fila: você pode usar exclusivamente vetores - o vetor de contextos e um vetor para indicar quais PIDs estão vivos (que devem ser escalonados em algum momento). Assuma que o PID do primeiro processo vale 1.

Para executar o seu escalonador, é preciso utilizar o simulador ARM. Para tanto, você deve:

1. Montar os arquivos do seu escalonador usando o utilitário `arm-eabi-as`;
2. Ligar os objetos gerados na etapa anterior com o `arm-eabi-ld` gerando um executável, por exemplo, chamado `raXXXXXX`. Note que para ligar seu código de sistema, você deve seguir o mesmo procedimento do Laboratório 8;
3. Montar e ligar o código do *DummyUSER*, gerando um executável denominado, por exemplo, `dummy_user`;
4. Criar uma imagem de cartão SD usando o utilitário `mksd.sh`, da seguinte forma: `mksd.sh --so raXXXXXX --user dummy_user`;
5. Executar a imagem do SD via simulador, usando o comando: `arm-sim --rom=dumboot.bin --sd=disk.img`.
6. Note que você pode usar o `arm-eabi-gdb` normalmente para realizar a depuração, conforme feito em laboratórios anteriores.

5 Entrega e Avaliação

O trabalho pode ser entregue nas modalidades individual ou em dupla, mas trabalhos em dupla podem ter arguição adicional. Caso haja qualquer tentativa de fraude, como plágio, todos os envolvidos receberão nota 0 na média da disciplina. O prazo de entrega do trabalho no sistema Susy é 23 de Dezembro de 2014. A entrega consistirá em:

- Código-fonte completo e comentado do escalonador, além de um arquivo `Makefile` que gere o executável como regra padrão; Todos esses arquivos devem ser comprimidos em um único arquivo `.tar.gz`, com o nome `raXXXXXX.tar.gz`, em que `XXXXXX` é o número de seu RA. O executável do simulador, que será gerado pelo `Makefile`, deve ser nomeado `raXXXXXX` (sem extensão mesmo!). Em caso de trabalho feito em dupla, utilize a convenção `raXXXXXX_raYYYYYY` tanto no arquivo comprimido quanto no executável do simulador e submeta o arquivo com a conta de um dos envolvidos;
- Programas de exemplo que demonstrem o funcionamento correto do escalonador.