EP2: MAC0422 - Sistemas Operacionais

Thiago Cunha Ferreira e Eduardo Freire de Carvalho Lima

1 Introdução

Este relatório descreve o que foi feito para o segundo EP (Exercício-Programa) da matéria *MAC0422 - Sistemas Operacionais*, onde tivemos que fazer modificações no sistema operacional para:

- Mostrar informações sobre os processos gerenciados pela escalonador quando pressionado a tecla F4 num terminal do MINIX.
- Implementar uma chamada de sistema ***chpriority(pid, priority)*** que muda a prioridade de processo-filho para uma prioridade válida.

IMPORTANTE: DURANTE A INICIALIZAÇÃO DO MINIX, NÃO SELECIONE AS OPÇÕES 1 OU 2 DE INICIALIZAÇÃO. ESPERE OU USE A OPÇÃO 3.

Trechos de códigos modificados estão marcados entre uma longa sequência do símbolo # em forma de comentários, geralmente ocupando uma linha inteira. Além disso, arquivos modificados/adicionados estão especificados neste documento.

OBS: Importante frizar que, embora no PDF para esse EP o nome da função de usuário ser *setpriority*, um novo nome foi sugerido pelo monitor, *chpriority*, por gerar conflitos em alguns projetos. Isso foi resolvido no tópico *EP2 - Nome Setpriority* do fórum de discussão do Paca, apesar de não ter sido formalizado no PDF. Portanto **utilize a função com o novo nome, chpriority**.

2 Nova função para a tecla F4

Nessa etapa, tivemos que mostrar os processos que são gerenciados pelo escalonador ao pressionarmos a tecla F4. Para isso, tivemos que acessar a pasta:

/usr/src/servers/is

onde ficaram localizados os três arquivos importantes para essa tarefa: o arquivo *dmp.c*, *proto.h* e o *dmp_kernel.c*. No *dmp.c*, tivemos que alterar a struct *hook_entry* para quando a tecla F4 fosse pressionada, a nossa função chamada de *ep_dmp* fosse executada. No *proto.h*, incluímos um _PROTOTYPE para nossa função. Ela recebe void como parâmetro e o possui também como retorno da função.

_PROTOTYPE(void ep_dmp, (void))

No *dmp_kernel.c*, criamos nossa função chamada de *ep_dmp*. Nos baseamos na função *proctable_dmp* para criá-la. Para isso, pegamos uma cópia do tabela de processos atual através da função *sysgetproctable*. Para imprimir os processos desejados, fizemos uso da struct *proc*, localizada em:

/usr/src/kernel/

A struct *proc*, chamada aqui de "rp", possui todos os campos desejados pela tarefa:

- Prioridade de execução: através do campo rp->p_priority;
- Process ID: através da função chamada *getnpid()*, cujo parâmetro é um endpoint(rp ->p_endpoint) e retorna o valor do pid.
- Tempo de CPU: rp->p_user_time;
- Tempo de Sistema: rp->p_sys_time;
- Endereço do ponteiro da pilha: rp->p_reg.sp;

Para imprimir os processos em ordem de prioridade, varremos a lista dos processos enquanto não fossem impressos o número de processos que gostaríamos (escolhemos o número 20). Nisso, começamos com prioridade 0 (vale lembrar que as prioridade vão de 0 a 15, sendo 0 a das task) e, durante a varredura, sempre que um processo tiver essa prioridade, ele será impresso e o contador de processos impressos será incrementado em 1. Isso é feito até completar todos os processos. Vale ressaltar que colocamos a condição de proc_nr(rp) >0 para que os processos-task não fossem impressos.

Ao final de cada varredura, verificamos se os 20 processos foram atingidos: caso não tenha sido, incrementamos a prioridade de 1 e voltamos ao começo da lista de processos. Caso tenha sido atingido, marcamos o último processo impresso para que comecemos do seguinte. Como há mais de 20 processos na tabela, 2 mensagens podem aparecer ao final do processo: "For more processes, press F4 again" ou "End of processes. Press F4 to show them again". A primeira mensagem aparece quando imprimimos os 20 processos com menor prioridade e, por existir mais, ele pede que aperte F4 novamente. Já a segunda mensagem aparece quando todos os processos foram impressos.

3 Chamada de sistema *chpriority*

Nessa etapa do EP, tivemos que implementar uma função que mudasse a prioridade de um processo-filho, especificado pelo argumento *pid*, para uma prioridade *"priority"* válida. Para tanto, essa etapa pode ser divida em 2 partes distintas, uma que trata da criação da chamada de sistema e outra da criação de uma chamada ao kernel.

Como essa etapa envolveu a modificação e inclusão de diversos arquivos, as duas partes seguintes serão construídas sobre as modificações e inserções de arquivos, bem como a "região" a que fazem parte (bibliotecas, servidores, kernel etc.). Quando um arquivo listado for prescindido de um símbolo "+", quer dizer que ele foi adicionado.

3.1 Chamada de sistema

Primeiramente, foi necessário implementar a função que o usuário deveria chamar, ou seja, a interface usuário-SO. Para isso, os arquivos relevantes foram:

- + */usr/src/lib/posix/_chpriority.c*

  Define a função *chpriority()*, que pode ser chamada pelo usuário, com dois parâmetros específicos: um *PID* de um processo e uma prioridade. Essa é a interface de comunicação entre usuário/*Process Manager*.

  Em caso de erro, os valores serão tanto retornados com a própria função quanto estarão armazenados na variável *errno*, definida em */usr/src/include/errno.h* e armazenada pela chamada de "_syscall()".

- */usr/src/lib/posix/Makefile.in*

  Adicionado arquivo _chpriority.c para que possa ser gerado um novo Makefile (ao executar o comando *make Makefile*), esse podendo ser utilizado por outras ferramentas de compilação do SO.

- */usr/src/include/unistd.h*

  Adicionado protótipo da chamada de sistema. Apesar ela poder ser usada mesmo sem incluí-la no projeto, possibilita a remoção das mensagens de *warning* do compilador *CC* em relação a função.

- */usr/src/include/minix/callnr.h* e */usr/include/minix/callnr.h*

  Adicionado o número da chamada de sistema para comunicação com o *Process Manager* (em */usr/src/servers/pm/table.c*), número 69, código NEWSETPRIORITY (macro SETPRIORITY já existente).

Após isso, temos os arquivos responsáveis por parte da implementação da função no *Process Manager*, associadas à função *do_setpriority()*:

- */usr/src/servers/pm/misc.c*

  Implementa a função *do_setpriority()* do *Process Manager*, que verifica relação de parentesco entre o processo que chamou a função *chpriority()* e o *pid* passado como argumento e chama um novo *kernel call* (detalhado na próxima subseção).

  A implementação dessa função foi baseada em uma outra função, *do_getsetpriority()*, presente no mesmo arquivo. Apesar do nome, há uma distinção no modo de execução delas, visto que essa muda a prioridade do processo através do valor *nice* do processo, enquanto nossa implementação muda a prioridade diretamente.

- */usr/src/servers/pm/proto.h*

  Adicionado protótipo da função *do_setpriority()* em */usr/src/servers/pm/misc.c*.

- */usr/src/servers/pm/table.c*

  Adicionado mapeamento entre número da chamada de sistema e sua implementação no *Process Manager*.

3.2 Chamada de kernel

Em uma das linhas de código da função *do_setpriority()* em */usr/src/servers/pm/misc.c*, há a chamada à função *sys_setpriority()*. Ela é a nova chamada ao kernel, que faz a intermediação entre o *Process Manager* e o kernel, através do *System Task*. Os arquivos relacionados são:

- + */usr/src/lib/syslib/sys_setpriority.c*

  Criada implementação da função que passa controle para o *System Task*, *sys_setpriority()*.

- */usr/src/include/minix/syslib.h*

  Adicionado protótipo da função acima.

- */usr/src/include/minix/com.h*

  Adicionado o número correspondente à implementação da função acima.

- */usr/src/lib/syslib/Makefile.in*

  Adicionado o arquivo da implementação da função acima para geração de novo Makefile e subsequente compilação.

Após isso, foi implementada a chamada ao kernel em si, relacionada com os arquivos:

- + */usr/src/kernel/system/do_setpriority.c*

  Implementação da chamada ao kernel com o nome *do_setpriority()* (mesmo nome da função que atual no *Process Manager*, mas agora no *kernel*), que efetivamente muda a prioridade do processo, checando se essa mudança é válida (verificando os limites impostos pela prioridade máxima e prioridade máxima de usuário).

  É interresante destacar que a checagem de prioridades é feita com macros já existentes no SO, MAX_USER_Q e MIN_USER_Q. Apesar de o intervalo entre elas incluir a fila de prioridade máxima 0, **excluímos a possibilidade de alteração da prioridade de um processo para a 0**, reservando-a para os *tasks*.

  OBS: Além da existência de macros que definem prioridade para processos de usuários, de task e IDLE, não foi encontrada nenhuma outra subdivisão entre as prioridades 1-14, tanto na implementação quanto na teoria de desenvolvimento do MINIX.

- */usr/src/kernel/system.h*

  Adicionado protótipo da função acima.

- */usr/src/kernel/system.c*

  Adicionado mapeamento entre *Process Manager-System Task* (com a definição do macro em */usr/src/include/minix/com.h*).

- */usr/src/kernel/config.h*

  Permite ativação da função *do_setpriority()* (está relacionado com a definição de seu protótipo em */usr/src/kernel/system.h*).

- */usr/src/kernel/system/Makefile*

  Adicionado arquivo */usr/src/kernel/system/do_setpriority.c* para ser compilado.

3.3 Informações adicionais

A compilação desse EP foi feito localmente em algumas partes e usando o Makefile presente em */usr/src*.

Além disso, nota-se que há código de erros adicionais nas duas funções *do_setpriority()* não especificados no enunciado. Eles servem para diferenciar problemas diferentes daqueles do PDF. Por exemplo, se for passado um PID inexistente, será retornado o valor -3, diferenciando do valor -2 que representa a inexistência da relação pai/filho com o PID especificado.