EXERCÍCIO-PROGRAMA 2: ESCALONAMENTO DE PROCESSOS

SISTEMAS OPERACIONAIS — MAC0422

RENATO LUI GEH
NUSP: 8536030
GUILHERME FREIRE
NUSP: 7557373

1. Introdução

O EP foi feito em um Minix 3.1.2a simulado pela VM VirtualBox. Os arquivos fonte estão localizados em `/usr/local/`.

Os arquivos modificados foram:

- `/usr/src/kernel/proc.c`
- `/usr/src/kernel/proc.h`
- `/usr/src/servers/pm/table.c`
- `/usr/src/servers/pm/misc.c`
- `/usr/src/servers/pm/proto.h`
- `/usr/src/lib/posix/Makefile.in`
- `/usr/src/include/minix/callnr.h`
- `/usr/include/minix/callnr.h`

As versões modificadas estão em `/usr/local/`, assim como os arquivos não modificados. Deste jeito, pode-se rodar `/usr/local/src/tools/Makefile` sem alterar o código original. Um arquivo foi adicionado:

- `/usr/local/src/lib/posix/_fork_batch.c`

Quando os blocos de código transcritos neste relatório não forem muito grandes, vamos indicar as modificações feitas. Um símbolo `-` no início da linha indica a linha original no Minix. Um símbolo `+` no início da linha indica a nova linha adaptada para o EP. Uma linha vazia com o símbolo `-` indica que no código original a linha não existia. Analogamente, `+` em uma linha vazia indica que deletamos a linha original. Um `#` indica um comentário no código, ou seja, a linha indicada por este símbolo não existe no arquivo original.

Date: 1 de outubro de 2016.

2. Kernel

As modificações no kernel foram feitas nos arquivos:

(1) `/usr/local/src/kernel/proc.c`
(2) `/usr/local/src/kernel/proc.h`

No arquivo 2, adicionamos uma nova macro `BATCH_Q`:

```
-
+ #define BATCH_Q 15 /* batch queue, before IDLE and
    after user queues. */
```

Esta nova macro indica uma nova fila de prioridade, antes da fila `idle` (`IDLE_Q`) e depois da última fila de prioridade de usuário (`MIN_USER_Q`). Como adicionamos uma nova fila, precisamos incrementar em um a macro que indica o número total de filas:

```
- #define NR_SCHED_QUEUES 16
+ #define NR_SCHED_QUEUES 17
```

Além disso, como estamos transladando a fila de `idle`, precisamos incrementa-la também.

```
- #define IDLE_Q 15
+ #define IDLE_Q 16
```

Estas modificações em `proc.h` concluem a tarefa 1 do EP. Agora discutiremos as mudanças feitas em `proc.c`, que coincidem justamente com a tarefa 3.

Seguindo a convenção ANSI descrita em `/usr/lib/ansi.h` e seguida pelo Minix 3.1.2a, devemos primeiro declarar a nova função de escalonamento em batch:

```
-
+ FORWARD _PROTOTYPE (void sched_batch, (struct proc *rp,
    int *queue, int *front));
```

Vamos agora transcrever e analisar a nova função `sched_batch` que trata do escalonamento em batch.

```
1  /*===========================================================================*
2   *        sched_batch          *
3   *===========================================================================*/
4
5  PRIVATE void sched_batch(rp, queue, front)
6  register struct proc *rp;  /* process to be scheduled */
7  int *queue;   /* return: queue to use */
8  int *front;   /* return: front or back */
9  {
10   register struct proc *batch_it;
11   int lmin, diff;
12
13   batch_it = rdy_head[BATCH_Q];
14   diff = 0;
15   lmin = -1;
16   if (batch_it->p_time_left <= 0) {
17     /* Find 'last' proc wrt user time */
18     for (; batch_it != NIL_PROC; batch_it = batch_it->p_nextready) {
19       if (batch_it == rp) continue;
20       if (lmin < 0)
21         lmin = batch_it->p_user_time;
22       else if (batch_it->p_user_time < lmin)
23         lmin = batch_it->p_user_time;
24     }
25     /* Invariant: diff >= 0, since lmin is minimum */
26     diff = rp->p_user_time - lmin;
27     /* If diff == 0, rp is next to lmin => rp must go front */
28     if (diff <= 0)
29       rp->p_ticks_left = rp->p_quantum_time;
30     /* Else, rp is 'in front' of lmin => rp must wait for last proc */
31     else
32       rp->p_ticks_left = 0;
33   }
34   *queue = BATCH_Q;
35   /* If there is still time left, keep it front. Else, depends on diff. */
36   *front = !diff;
37 }
```

A função `sched_batch`, assim como a função original `sched` do Minix, toma como argumentos uma `struct proc*` que representa o endereço do processo a ser escalonado, e dois endereços para inteiros, `queue` que sinaliza qual a fila de prioridade para se usar, e `front` que indica se o processo deve ir na frente da fila ou atrás.

Vamos analisar a função. Antes de mais nada, declaramos as variáveis que iremos utilizar. A variável `register struct proc *batch_it` será, no `for` da linha 18, o endereço para processo de cada item da fila de prioridade indexado por `BATCH_Q`.

O inteiro `lmin` é o menor tempo de usuário de todos os processos da fila. Já `diff` será usado para medir a diferença entre o menor tempo de usuário e o tempo de usuário do processo a ser escalonado.

As linhas 18-24 apenas acham o menor tempo de execução da fila de processos em batch. Percorremos a fila e consideramos os tempos de execução de usuário desde que o processo visto é distinto daquele que estamos escalonando. Em seguida, na linha 26, atribuímos o valor da diferença entre o menor tempo ao tempo de execução de usuário do processo a ser escalonado. Note a invariância de que nesta linha `diff` ≥ 0. Isso ocorre pois, como `lmin` é mínimo, então se considerarmos o oposto, então `rp->p_user_time < lmin`, o que é uma contradição.

Note que na linha 14, inicializamos a variável `diff` como 0. Isso ocorre pois devemos considerar dois casos. No caso em que o processo ainda possue tempo de execução (ou seja, se a linha 16 retornar falso) devemos colocar o processo no começo da fila. Como diff é inicializado como 0, a linha 36 funciona como intencionado. Agora considere o caso em que o processo precisa ser re-escalonado (ou seja, se a linha 16 retornar verdadeiro). Neste caso, vamos ter dois subcasos:

1o. subcaso

1.1. Temos que `diff == 0` (linha 28).
1.2. Ou seja, `rp` está tão "atrasado" quanto o processo que rodou menos.
1.3. Isto indica que devemos rodar `rp` antes, já que ele está empatado com o último.
1.4. Portanto, `rp` deve ir "na frente" da fila. Como `diff` é 0, `*front=1`.
1.5. Além disso, damos um tempo de ticks igual ao número de ticks equivalente a um quantum.

2o. subcaso

2.1. No segundo caso, `diff > 0` (linha 31).
2.2. Ou seja, `rp` está "adiantado" em relação ao processo mais "atrasado".
2.3. Isto indica que devemos rodar o processo mais atrasado antes de `rp`.
2.4. Portanto, `rp` deve ir no final da fila e dar passagem para os processos atrasados, e já que `diff`> 0, então `*front=0`.
2.5. Como vamos "pular" este processo, damos um tempo de 0 ticks restantes e pomos no final da fila.

Perceba que, quando todos os processos "empatarem" em relação ao tempo de execução, um deles será escolhido e dado um tempo equivalente a um quantum de tempo. Quando este terminar, todos os outros processos também percorrerão um quantum de tempo cada um, um de cada vez. Além disso, todos os processos que estão na frente esperarão os atrasados. Isso é a definição de escalonamento *Round Robin*, como foi pedido no enunciado quando todos os processos empatam.

Ao final da função, anunciamos que a fila de prioridade a ser usada é aquela indexada por `BATCH_Q` (linha 34). Em seguida, atribuímos o valor de `diff` como descrito anteriormente.

Agora resta chamarmos a função de escalonamento. Faremos isso dentro da função `enqueue`, no próprio `proc.c`.

```
1  PRIVATE void enqueue(rp)
2  register struct proc *rp; /* this process is now runnable */
3  {
4    /* ... */
5    /* Determine where to insert to process. */
6      /* ############################################################ */
7      if (rp->p_priority == BATCH_Q)
8        sched_batch(rp, &q, &front);
9      else
10     /* ############################################################ */
11       sched(rp, &q, &front);
12   /* ... */
13 }
```

Quando o processo a ser escalonado tem prioridade igual a `BATCH_Q`, escalonamos com a função `sched_batch`, senão escalonamos normalmente com `sched`.

3. Preparando o `syscall`

Para criarmos a *system call*, vamos primeiro alterar os arquivos do servidor responsável pelo gerenciamento de processos (`process manager`), que se encontram em

```
/usr/src/servers/pm/
```

Primeiro, encontramos no arquivo `table.c` um endereço que não está sendo utilizado para alocar a nossa função. O endereço escolhido é o 57, então temos:

```
-   no_sys, /* 57 = unused */
+   /* ######################################## */
+   do_fork_batch,     /* 57 = FORK_BATCH */
+   /* ######################################## */
```

Nossa função está implementada dentro do arquivo `misc.c` (linha 31), e declarada no arquivo `proto.h` (linha 60):

```
-
+ _PROTOTYPE(int do_fork_batch, (void));
```

Modificado esses três arquivos, o próximo passo foi compilar o Process Manager, utilizando o Makefile dentro de `/usr/local/src/servers/`

```
cd /usr/local/src/servers/
make image
make install
```

Nesse ponto temos a função implementada, mas não explícita para o usuário. Para isso, primeiro definimos sua constante nos arquivos `usr/local/include/minix/callnr.h` e `/usr/local/include/minix/callnr.h`

```
-
+ /* ######################################### */
+ #define FORK_BATCH        57
+ /* ######################################### */
```

Adicionamos também, uma função que encapsula a syscall, facilitando para o usuário. Ela está contida num arquivo novo:

```
/usr/local/src/lib/posix/_fork_batch.c
```

Precisamos inserir esse arquivo novo no Makefile, para poder ser compilado com a biblioteca, então em `/usr/local/src/lib/posix/Makefile.in` adicionamos (linha 32),

```
_fork_batch.c\
```

e depois executamos

```
make Makefile
```

É possível então fazer a nova biblioteca com:

```
cd /usr/local/src
make libraries
```

Agora é possível fazer a chamada de sistema utilizando o comando `fork_batch()`. Para salvar essa nova configuração, precisamos criar uma imagem do sistema. Assim é possível dar boot nela em outras sessões. O comando é:

```
cd /usr/local/src/tools
make hdboot
make install
```

A imagem criada fica em `/boot/image/`.

4. Sincronizando as tabelas

Como modificamos as tabelas em `pm`, também devemos modificar as tabelas em `fs`. Para isso vamos modificar os arquivos em `/usr/local/src/servers/fs/`:

```
# /usr/local/src/servers/fs/table.c
- no_sys,  /* 57 = unused */
+ /* ######################################### */
+ do_fork_batch,    /* 57 = FORK_BATCH */
+ /* ######################################### */
```

```
# /usr/local/src/servers/fs/proto.h
-
+ /* ########################################### */
+ do_fork_batch,     /* 57 = FORK_BATCH */
+ _PROTOTYPE(int do_fork_batch, (void));
+ /* ########################################### */
```

E para `/usr/local/src/servers/fs/misc.c`:

```
/* ################################################ */
/*==================================================*
 *        do_fork_batch            *
 *==================================================*/
PUBLIC int do_fork_batch()
{
  return do_fork();
}
/* ################################################ */
```

Como `/usr/local/src/servers/pm/misc.c` chama `do_fork`, precisamos tratar o caso de `do_fork` no `fs`. Para isso apenas chamamos `do_fork` do fs e retornamos o valor.

5. DO_FORK_BATCH

Agora resta implementar o `fork_batch`. Para tal, iremos modificar o arquivo

```
/usr/local/src/servers/pm/misc.c
```

A escolha de termos utilizado `misc.c` ao invés de criado um novo arquivo `forkbatch.c` foi por causa da maior facilidade: `misc.c` já inclue `usr/local/src/kernel/proc.h` em seu código, assim como `usr/local/include/minix/callnr.h`, além de já ser um `target` no Makefile do `pm`.

A função que fará o `fork batch` será chamada `do_fork_batch`, assim como foi mencionado na seção anterior. A função não receberá argumentos e retornará um inteiro que indica o `pid` (*process id*) do processo filho se a chamada for bem sucedida. Caso contrário, a função retorna o erro:

- `EAGAIN`: a tabela de processos está cheia.
- `ENOMEM`: não há memória suficiente.

Vamos analizar a função:

```
1  PUBLIC int do_fork_batch()
2  {
3    int proc_id;
4
5    proc_id = do_fork();
6    if (proc_id == EAGAIN || proc_id == ENOMEM)
7      return proc_id;
8    sys_nice(proc_id, BATCH_Q);
9
10   return proc_id;
11 }
```

Note que na linha 5 fazemos uma chamada para `do_fork`. Como já temos uma chamada em `/usr/local/src/servers/pm/forkexit.c` que faz exatamente o fork de um processo, vamos reutilizar a chamada.

Em seguida vamos verificar a saída da chamada `do_fork`. Esta função retorna o ID do processo caso bem sucedida ou um erro caso falhe. No caso, a função apenas retorna `EAGAIN` ou `ENOMEM`, portanto verificamos na linha 6 por estes erros.

Na linha seguinte chamamos `sys_nice`. Esta função aceita como argumentos o número do processo (ou seja, o `pid`) e a prioridade, respectivamente. Temos o `pid` da chamada `do_fork`. Só nos resta a prioridade. Mas como `misc.c` já inclue o header `/usr/local/src/kernel/proc.h`, podemos usar a macro definida na Seção 1: `BATCH_Q`.

Finalmente, retornamos o processo que acabamos de criar.

6. Compilando

Para compilarmos tudo, rodamos os seguintes comandos:

```
cd /usr/local/src/tools
make clean install
```

Isto cria uma imagem

```
/usr/local/src/tools/image
```

Que podemos colocar em `/boot/image/` e rodarmos com:

```
cp /usr/local/src/tools/image /boot/image/batch_image
shutdown
image=/boot/image/batch_image
boot
```