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

Date: 29 de outubro de 2016.

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. Três arquivos foram adicionados:

- `/usr/local/src/lib/posix/_fork_batch.c`
- `/usr/local/src/lib/syslib/sys_fork_batch.c`
- `/usr/local/src/kernel/system/do_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.

2. Escalonamento em batch

Para fazer o escalonamento em batch foram modificados os seguintes arquivos do kernel:

(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.

```
/*===========================================================================*
 *        sched_batch          *
 *===========================================================================*/

PRIVATE void sched_batch(rp, queue, front)
register struct proc *rp;  /* process to be scheduled */
int *queue;   /* return: queue to use */
int *front;   /* return: front or back */
{
  register struct proc *batch_it;
  int lmin, diff;

  batch_it = rdy_head[BATCH_Q];
  diff = 0;
  lmin = -1;
  if (batch_it->p_time_left <= 0) {
    /* Find 'last' proc wrt user time */
    for (; batch_it != NIL_PROC; batch_it = batch_it->p_nextready) {
      if (batch_it == rp) continue;
      if (lmin < 0)
        lmin = batch_it->p_user_time;
      else if (batch_it->p_user_time < lmin)
        lmin = batch_it->p_user_time;
    }
    /* Invariant: diff >= 0, since lmin is minimum */
    diff = rp->p_user_time - lmin;
    /* If diff == 0, rp is next to lmin => rp must go front */
    if (diff <= 0)
      rp->p_ticks_left = rp->p_quantum_time;
    /* Else, rp is 'in front' of lmin => rp must wait for last proc */
    else
      rp->p_ticks_left = 0;
  }
  *queue = BATCH_Q;
  /* If there is still time left, keep it front. Else, depends on diff. */
  *front = !diff;
}
```

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. Kernel call

Para criarmos a *system call* `fork_batch`, precisamos criar uma *kernel call* (no caso chamada de `sys_fork_batch`) que irá criar o novo processo e dar a prioridade de batch ao processo filho. Em seguida podemos criar uma *system call* para mandarmos uma mensagem para o system task executar `sys_fork_batch`. Nesta seção trataremos da kernel call. Na seção seguinte iremos enunciar o *system call* e finalmente, na última seção sobre a implementação, iremos mostrar como foi feita a chamada para usuário.

Para criarmos o *kernel call*, devemos avisar ao kernel que adicionamos uma syscall. Em `minix/com.h`, incrementamos o número de syscalls:

```
- #define NR_SYS_CALLS 31
+ #define NR_SYS_CALLS 32
```

E em seguida adicionamos uma nova macro sinalizando a nova kernel call:

```
-
+ #define SYS_FORK_BATCH (KERNEL CALL + 31) /*
    sys_fork_batch() */
```

Agora podemos criar uma nova `sys_function` em `.../src/kernel/system/`. Mas antes vamos declarar o protótipo da função em `.../include/minix/syslib.h`:

```
-
+ _PROTOTYPE(int sys_fork_batch, (int parent, int child,
    int*));
```

Agora devemos criar o handler para o `_taskcall`. Criamos o arquivo `.../src/lib/syslib/sys_fork_batch.c`.

```
#include ''syslib.h''

PUBLIC int sys_fork_batch(parent, child, child_endpoint)
int parent;
int child;
int *child_endpoint;
{
  /* A process has forked. Tell the kernel. */
  message m;
  int r;

  m.PR_ENDPT = parent;
  m.PR_SLOT = child;
  r = _taskcall(SYSTASK, SYS_FORK_BATCH, &m);
  *child_endpoint = m.PR_ENDPT;
  return r;
}
```

Compare com o arquivo `.../src/lib/syslib/sys_fork.c`. Ambos são iguais com a exceção do `_taskcall`. Isso pois ambos criam um processo, porém `fork_batch` irá atribuir uma prioridade fixa ao processo.

Agora podemos implementar o nosso `fork_batch` no kernel. Para isso, criamos um arquivo `.../src/kernel/system/do_fork_batch.c`. O arquivo é igual ao seu `fork` equivalente, porém com a seguinte linha adicional:

```
rpc->p_sys_time = 0;
-
+ rpc->p_priority = rpc->p_max_priority = BATCH_Q;
```

Ou seja, daremos uma prioridade fixa `BATCH_Q`, que corresponde exatamente a nossa fila de prioridade `batch` enunciada na Seção 2. Agora precisamos avisar ao `.../src/kernel/system.h` que implementamos tal função:

```
-
+ _PROTOTYPE(int do_fork_batch, (message *m_ptr));
+ #if !USE_FORK_BATCH
```

```
+ #define do_fork_batch do_unused
+ #endif
```

Agora temos nossa kernel call, que será chamada a partir de um system call.

4. System call

Para criarmos a *system call*, vamos primeiro alterar os arquivos do servidor responsável pelo process manager (pm) e do file system (fs), que se encontram em

```
/usr/local/src/servers/pm
/usr/local/src/servers/fs
```

Primeiro, encontramos no arquivo .../src/servers/pm/table.c um endereço que não está sendo utilizado para alocar a nossa função. Também fazemos a mesma modificação em .../src/servers/fs/table.c. O endereço escolhido é o 57, então temos:

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

Em seguida declaramos o protótipo da função em .../src/servers/pm/proto.h e em seu equivalente no fs:

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

Implementamos a função em .../src/servers/pm/misc.c. Toda a função age igual ao seu equivalente do_fork em .../src/servers/pm/forkexit.c com a exceção da linha:

```
-
+ rmc->mp_nice = BATCH_Q;
```

Que diz ao mproc que a prioridade é BATCH_Q. E também os dois seguintes trechos:

```
# Chama fork_batch ao inves de fork.
- if((r=sys_fork(who_e, child_nr, &rmc->mp_endpoint)) !=
    OK)
+ if((r=sys_fork_batch(who_e, child_nr, &rmc->mp_endpoint
    )) != OK)
# Diz ao SO para atualizar o fs do novo processo rodando
    a funcao do FORK_BATCH em fs.
- tell_fs(FORK, who_e, rmc->mp_endpoint, rmc->mp_pid);
```

```
+ tell_fs(FORK_BATCH, who_e, rmc->mp_endpoint, rmc->
    mp_pid);
```

Para o equivalente do `fs`, apenas chamamos `do_fork`, já que a atualização das tabelas é idêntico ao `fork` normal. Agora devemos dizer ao SO qual é o endereço da chamada de sistema. Em `...minix/callnr.h`:

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

Isso indica que o syscall 57 é definido pela chamada `FORK_BATCH`, que mapeia a função `do_fork_batch` no `callvec` da `table.c` modificada anteriormente.

5. Library handler

Agora que temos as syscalls e kernel calls, podemos chamar a função `fork_batch` por meio da `_syscall`:

```
_syscall(PM_PROC_NR, FORK_BATCH, &m);
```

Onde `m` é uma variável do tipo `message`. Mas a função `_syscall` não deve ser chamada pelo usuário diretamente. Para resolvermos esse problema, criaremos uma função de usuário que chama o syscall.

Modificaremos o arquivo `.../include/unistd.h`:

```
-
+ _PROTOTYPE(pid_t fork_batch, (void));
```

E em seguida implementaremos a função em `/usr/local/src/lib/posix/_fork_bath.c` que fará a chamada `_syscall` pelo usuário:

```c
#include <lib.h>
#include <unistd.h>

PUBLIC pid_t fork_batch()
{
  message m;
  return _syscall(MM, FORK_BATCH, &m);
}
```

Note que `MM` é definido em `lib.h` como:

```
#define MM PM_PROC_NR
```

Portanto, no final `fork_batch` apenas servirá de *wrapper* para a chamada de função `_syscall`.

6. Usando `fork_batch`

Agora podemos chamar a nossa nova função `fork_batch` da seguinte forma:

```
pid_t chid;
chid = fork_batch();
if (chid == 0) {
  /* Processo pai */
} else if (chid == EAGAIN || chid == ENOMEM) {
  if (chid == EAGAIN) {
    /* Tabela de processos cheia */
  } else {
    /* Sem memoria */
  }
} else {
  /* Processo filho */
}
```

7. Compilando

Para compilarmos tudo, rodamos os seguintes comandos:

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

Isto cria uma nova imagem em `/boot/image/` com a convenção:

```
3.1.2arN
```

Onde `N` é a versão da imagem. Para rodarmos a imagem para aplicarmos as modificações feitas no código:

```
shutdown
image=/boot/image/3.1.2arN
boot
```

8. Observações importantes quanto a execução do EP

Como usamos um Floppy Controller na nossa VM, a tela inicial irá indicar que não foi encontrado um local de boot. Para resolver isto, pressione `F12` e em seguida pressione `1`. Isto selecionará o controlador principal (a que possue a imagem do Minix) como local de boot.

Como não mudamos o caminho de boot padrão do Minix, quando a VM for rodada, deve-se dar boot na imagem correta. Por padrão, a VM irá dar boot na imagem padrão original.

É recomendável que se recompile o Minix novamente para garantir que tudo esteja o mais recente possível. Caso não se recompile o Minix, a imagem em `/boot/image` mais recente é:

```
/boot/image/3.1.2ar30
```

Para rodar a imagem escolhida, basta indicar o caminho. Por exemplo, caso a imagem desejada seja `/boot/image/3.1.2ar30`, então:

```
# Garanta que esteja na tela de boot.
shutdown
# Indique qual imagem deve ser escolhida.
image=/boot/image/3.1.2ar30
# Faca o boot.
boot
```

9. Testes

Fizemos alguns testes que se encontram no diretório `/home/ep2`. Apenas compile e rode. Para executar o teste com o `fork` normal, não use argumentos. Para executar o teste como `fork_batch`, adicione um argumento.

```
# Rodando com fork normal:
cc test.c
./a.out
# Rodando com fork_batch:
cc test.c
./a.out 0
```