Exercício Programa 2 – Relatório

MAC0422 – Sistemas Operacionais

André Spanguero Kanayama 7156873
Pedro Paulo Vezzá Campos 7538743

5 de novembro de 2013

1. Enunciado

Para este segundo exercício-programa de MAC0422 – Sistemas Operacionais, o professor requisitou que os alunos alterassem o *Process Manager* (PM) para implementar uma biblioteca de semáforos.

2. Desenvolvimento da implementação

Após consultas a tutoriais na Internet, encontramos os principais arquivos necessários para a implementação de uma biblioteca C que se comunique com o PM:

/usr/src/servers/pm/semaphore.c Local da implementação efetiva dos semáforos no PM

/usr/src/lib/libc/posix/_semaphore.c Local do código da biblioteca de usuário responsável por abstrair as chamadas de sistema feitas através de mensagens no Minix.

/usr/src/include/minix/callnr.h Arquivo de macros para os códigos definidos para cada uma das chamadas de sistema do Minix.

/usr/src/servers/pm/table.c Define a tabela que mapeia um código de chamada de sistema definido no arquivo `callnr.h` para uma função a ser executada pelo PM.

/usr/src/servers/pm/proto.h Arquivo de protótipos das funções do PM.

2.1. Código da biblioteca

O código da biblioteca de usuário é bastante simples. Se resume a montar uma mensagem a ser enviada para o PM contendo os parâmetros necessários, realizar a chamada à função `_syscall()` e aguardar uma mensagem de retorno contendo o resultado da chamada.

É neste ponto que o processo de usuário pode ser terminado e um código de erro -1 é enviado no caso de um usuário tentar criar um semáforo no momento que o sistema está com todos os 128 semáforos alocados. Isto foi implementado na função `get_sem()` analisando o código de retorno da chamada de sistema. Em caso de erro a biblioteca chama a função `exit()`.

É importante ressaltar que o bloqueio dos processos que fizerem uma operação "P" em um semáforo que está com valor atual 0 é feita através do sistema de troca de mensagens do Minix. O bloqueio acontece na chamada à função `_syscall()` dentro da função `p_sem()`

```
1 #include <sys/cdefs.h>
2 #include "../other/namespace.h"
3 #include <lib.h>
4 #include <unistd.h>
5 #include <stdlib.h>
6 #include <stdio.h>
7 #include <minix/callnr.h>
8
9 PUBLIC int get_sem (int valor) {
10    message m;
11    int result;
12    m.m1_i1= valor;
13    result = _syscall (PM_PROC_NR, GETSEM, &m);
14    if(result == 0){
15       exit(-1);
16    }
17    return result;
18 }
19
20 PUBLIC int v_sem (int indice_sem) {
21    message m;
22    m.m1_i1 = indice_sem;
23    return (_syscall (PM_PROC_NR, VSEM, &m));
24 }
25
26 PUBLIC int p_sem (int indice_sem) {
27    message m;
28    m.m1_i1 = indice_sem;
29    return (_syscall (PM_PROC_NR, PSEM, &m));
30 }
31
32 PUBLIC int free_sem (int indice_sem) {
33    message m;
34    m.m1_i1 = indice_sem;
35    return (_syscall (PM_PROC_NR, FREESEM, &m));
36 }
```

2.2. Modelagem de um semáforo

O arquivo `/usr/src/servers/pm/semaphore.c` contém a implementação verdadeira das funções requisitadas no enunciado.

Um semáforo foi modelado como uma `struct` C com a seguinte defínção:

```
1 struct semaforo {
2    int id, valor;
3    int fila_pid[TAM_FILA];
4    int begin;
5    int end;
6    int pcount;
7    int dono;
8 } semaforos[NR_SEMS];
```

A macro `NR_SEMS` foi definida como 128, conforme enunciado. A utilidade de cada componente está descrita a seguir:

id É uma ID criada aleatoriamente pela biblioteca e na faixa $[1, 1000000]$

valor Indica quantos processos podem entrar na região crítica no momento

fila_pid É uma fila implementada usando um vetor C que indica quais processos (Representados pelas suas posições na tabela de processos do PM) estão bloqueados aguardando liberação para entrar na região crítica.

begin, end Apontam respectivamente o início e o fim da fila no vetor `fila_pid`.

pcount Número de processos que estão aguardando liberação para entrar na região crítica.

dono Posição na tabela de processos do processo criador do semáforo.

2.3. Implementação das operações de manipulação de um semáforo

O arquivo `semaphore.c` contém as seguintes funções implementadas:

void init_sems (void) Invocada na primeira vez que é requisitado um semáforo.

void desaloca_pid (int who_p) Recebe como parâmetro a posição na tabela de processos de um processo que terminou a sua execução. Varre toda a tabela de semáforos e desaloca qualquer semáforo que tenha sido alocado a este processo.

int sem_exists (int sem) Varre a tabela de semáforos e retorna 1 caso a ID passada esteja presente e 0 caso contrário.

int authorized (int sem_index) Dada uma posição na tabela de semáforos, retorna 1 caso o processo que fez a requisição à biblioteca ou algum de seus ancestrais seja dono do semáforo referenciado.

int do_getsem (void) Tenta criar um novo semáforo para o processo chamador. Retorna a ID do novo semáforo na mensagem de retorno ou -1 em caso de erro. O código que é responsável por terminar um processo que requisitou um semáforo quando todos já estavam alocados é responsabilidade da biblioteca de usuário, como explicado anteriormente.

int do_vsem (void) Aplica a operação "V" no semáforo passado como parâmetro via o primeiro campo da mensagem do Minix. Caso haja algum processo bloqueado, libera-o, caso contrário, incrementa o valor atual do semáforo. Retorna -1 via mensagem caso a ID do semáforo seja inválida ou o usuário não esteja autorizado de acordo com a função `authorized()`.

int do_psem (void) Aplica a operação "P" no semáforo passado como parâmetro via o primeiro campo da mensagem do Minix. Caso o valor atual do semáforo seja 0, bloqueia-o, caso contrário, decrementa o valor atual do semáforo e libera o processo. Retorna -1 via mensagem caso a ID do semáforo seja inválida ou o usuário não esteja autorizado de acordo com a função `authorized()`.

int do_freesem (void) Libera o semáforo passado como parâmetro no primeiro campo da mensagem do Minix apenas se o remetente da mensagem for o dono do semáforo. Retorna 0 em caso de sucesso e -1 caso o rementente não esteja autorizado a realizar a operação ou a ID do semáforo não seja válida.

2.4. Desalocação de semáforos após o fim de um processo

Para satisfazer à restrição de desalocar todos os semáforos de um processo quando ele terminar a sua execução, vasculhamos o código fonte do PM até encontrar a função `exit_proc()` dentro do arquivo `/usr/src/servers/pm/forkexit.c`. Esta função é responsável por desalocar todas as estruturas utilizadas pelo PM no controle da execução de um processo.

A edição feita foi incluir mais uma linha nesta função. Nela, é feita uma chamada à função `desaloca_pid()`, responsável por desalocar todos os semáforos de um processo, como descrito anteriormente. O `diff` resultante é:

```
1    if (dump_core && (rmp->mp_flags & PRIV_PROC))
2    dump_core = FALSE;
3
4    proc_nr = (int) (rmp - mproc);  /* get process slot number */
5    proc_nr_e = rmp->mp_endpoint;
6  > /*????????????????????????????????????????????????????*/
7  > /*????????????????????????????????????????????????????*/
8  >   desaloca_pid(proc_nr);
9  > /*????????????????????????????????????????????????????*/
10 > /*????????????????????????????????????????????????????*/
11
12   /* Remember a session leader's process group. */
13   procgrp = (rmp->mp_pid == mp->mp_procgrp) ? mp->mp_procgrp : 0;
```

3. Dificuldades enfrentadas

Perdemos bastante tempo tentando entender o porquê de mesmo após alterarmos e recompilarmos o código da biblioteca de usuário, o comportamento do nosso programa de teste não mudava para o esperado. Concluímos que a "culpada" foi a ligação estática

de bibliotecas ao binário de teste compilado. A implementação da biblioteca de usuário é copiada para o binário final apenas durante a compilação do binário do usuário. Quando recompilamos nosso código de teste o comportamento passou a ser o esperado.

Na implementação da biblioteca de semáforos, vimos o PM utilizar a variável global `who_p` para referenciar o remetente da mensagem que está sendo processada. Acreditamos que essa variável fosse o PID do processo chamador mas na verdade `who_p` é a posição na tabela de processos referente ao remetente.

A. Implementação final da biblioteca

```
1  /************************************
2   *
3   * semaphore.c
4   *
5   ************************************/
6  #include "pm.h"
7  #include "param.h"
8  #include "glo.h"
9  #include "mproc.h"
10 #include <sys/wait.h>
11 #include <assert.h>
12 #include <signal.h>
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <string.h>
16 #include <sys/types.h>
17 #include <signal.h>
18
19 /* Numero maximo de semaforos */
20 #define NR_SEMS       128
21 #define TAM_FILA      1000
22 #define NO_SEM       -1
23 #define INIT_SEMVAL   1000
24 #define MAX_SEMVAL    1000000
25 #define INT_LIMIT     0x7fffffff
26 #define ESEMVAL       0x8000000
27 /*Ultrapassou o limite de semaforos*/
28 #define EFULL         0xDEADDEAD
29 #define NADA    0
30 /*Erro, retorna 0*/
31 #define RETERR        -1
32
33
34 FORWARD _PROTOTYPE( int sem_exists, (int sem)
                                    );
35 FORWARD _PROTOTYPE( void init_sems, (void)
                                      );
36
37 /*Definicao da estrutura do semaforo*/
38 struct semaforo {
39    int id, valor;
```

```
40      int fila_pid[TAM_FILA];
41      int begin;
42      int end;
43      int pcount;
44      int dono;
45  } semaforos[NR_SEMS];
46  int qtd_semaforos_ativos = 0;
47
48  /*Funcao executada uma vez, quando ainda nao existe nenhum semaforo*/
49  PRIVATE void init_sems (void) {
50      int i;
51
52      for (i = 0; i < NR_SEMS; i++) {
53          semaforos[i].id = NO_SEM;
54          semaforos[i].valor = 0;
55          semaforos[i].pcount = 0;
56          semaforos[i].fila_pid[0] = NADA;
57          semaforos[i].begin = 0;
58          semaforos[i].end = 0;
59      }
60      qtd_semaforos_ativos = 0;
61  }
62
63  /*Funcao usada pelo PM, para que quando um processo morra, todos os
        seus semaforos sejam liberados*/
64  PUBLIC void desaloca_pid (who_p)
65      int who_p;
66  {
67      int i;
68      for (i = 0; i < NR_SEMS; i++) {
69        if (semaforos[i].dono == who_p){
70            qtd_semaforos_ativos--;
71
72            semaforos[i].id = NO_SEM;
73            semaforos[i].valor = 0;
74
75            semaforos[i].pcount = 0;
76            semaforos[i].fila_pid[0] = NADA;
77            semaforos[i].begin = 0;
78            semaforos[i].pcount = 0;
79
80        }
81      }
82
83
84  }
85
86  /*Funcao que verifica se um semaforo ja existe*/
87  PRIVATE int sem_exists (sem)
88      int sem;
89  {
90      int i;
91      if (sem <= 0 || sem > INT_LIMIT - 1) return 0;
92
```

```
93      if (qtd_semaforos_ativos < 1) return 0;
94      for (i = 0; i < NR_SEMS; i++) {
95          /*Se encontrar uma id igual a passada, retorna 1*/
96          if (semaforos[i].id == sem) return 1;
97      }
98      return 0;
99  }
100
101 /*Funcao que verifica se o processo pode fazer uma operacao com o
        semaforo*/
102 PRIVATE int authorized (sem_index)
103     int sem_index;
104 {
105     int i;
106     int mproc_index = -1;
107     int pid_atual = who_p;
108     register struct mproc *atual;
109     atual = &mproc[pid_atual];
110
111     while(atual->mp_parent != pid_atual && pid_atual > 0 && semaforos[
            sem_index].dono != pid_atual){
112         /*Verifica se os processos "pais" sao donos do semaforo, e se um
                deles for, o processo eh autorizado*/
113         pid_atual = atual->mp_parent;
114         atual = &mproc[pid_atual];
115
116     }
117
118     if(semaforos[sem_index].dono == pid_atual)
119         return 1;
120     return 0;
121 }
122
123 /**********************************************GET SEM
        ****************************************************************/
124 PUBLIC int do_getsem (void) {
125     int resultado = RETERR;
126     int id_semaforo;
127     /*mensagem*/
128     int valor = m_in.m1_i1;
129     int i;
130
131     if (qtd_semaforos_ativos == 0) init_sems ();
132
133
134     if (qtd_semaforos_ativos >= NR_SEMS) {
135         m_in.m_type = EFULL;
136         return RETERR;
137     }
138
139     for (i = 0; i < NR_SEMS && resultado == RETERR; ++i) {
140         if (semaforos[i].id == NO_SEM) {
141             /*Gera um id aleatorio para o semaforo*/
142             id_semaforo = rand() % MAX_SEMVAL + 1;
```

```
143             /*Se semaforo ja existe, soma 100 no id ateh achar um numero
                    que nao exista*/
144             while (sem_exists (id_semaforo)) id_semaforo += 100;
145             qtd_semaforos_ativos++;
146             resultado = id_semaforo;
147 
148             semaforos[i].id = id_semaforo;
149             semaforos[i].valor = valor;
150 
151             semaforos[i].pcount = 0;
152             semaforos[i].fila_pid[0] = NADA;
153             semaforos[i].begin = 0;
154             semaforos[i].end = 0;
155             semaforos[i].dono = who_p;
156         }
157     }
158 
159     return resultado;
160 }
161 
162 
163 /*************************************************V SEM
        ***************************************************/
164 PUBLIC int do_vsem (void) {
165 
166     int resultado = RETERR;
167 
168     int sem = m_in.m1_i1;
169 
170     int i;
171     int index;
172 
173     if (sem <= 0 || sem > INT_LIMIT - 1 || qtd_semaforos_ativos < 1) {
174         m_in.m_type = EINVAL;
175         return RETERR;
176     }
177 
178     index = -1;
179     /*Procura o indice que corresponde ao semaforo no vetor de
            semaforos*/
180     for (i = 0; i < NR_SEMS; i++){
181         if(semaforos[i].id == sem){
182             index = i;
183             break;
184         }
185     }
186 
187     /*Nao achou o semaforo*/
188     if(index == -1)
189         return RETERR;
190 
191     /*Se nao for dono ou filho do dono nao pode fazer a operacao*/
192     if(!authorized(index))
193         return RETERR;
```

```
194
195     if (semaforos[index].valor == MAX_SEMVAL) {
196        m_in.m_type = EBUSY;
197        return RETERR;
198     }
199
200     (semaforos[index].valor)++;
201
202     if (semaforos[index].pcount != 0) {
203        int next = semaforos[index].begin;
204        int proc_nr = semaforos[index].fila_pid[next];
205
206        if (proc_nr <= NADA || proc_nr >= NR_PROCS) return RETERR;
207
208
209        if (next == semaforos[index].end) {
210           semaforos[index].fila_pid[next] = NADA;
211           semaforos[index].begin = 0;
212           semaforos[index].end = 0;
213        }else if (next == (TAM_FILA - 1)) {
214           semaforos[index].fila_pid[TAM_FILA - 1] = NADA;
215           semaforos[index].begin = 0;
216        }else {
217           semaforos[index].fila_pid[next] = NADA;
218           next++;
219           semaforos[index].begin = next;
220        }
221
222        (semaforos[index].pcount)--;
223        resultado = 0;
224
225        setreply (proc_nr, resultado);
226     }
227
228     return resultado;
229 }
230
231 /*********************************************************P SEM
        ************************************************************/
232 PUBLIC int do_psem (void) {
233
234     int resultado = RETERR;
235
236     int sem = m_in.m1_i1;
237
238     int index;
239
240     int i;
241
242     if (sem <= 0 || sem > INT_LIMIT - 1 || qtd_semaforos_ativos < 1) {
243        m_in.m_type = EINVAL;
244        return RETERR;
245     }
246
```

```
247     index = -1;
248     /*Procura o indice do semaforo no vetor de semaforos*/
249     for (i = 0; i < NR_SEMS; i++){
250        if(semaforos[i].id == sem){
251            index = i;
252            break;
253        }
254     }
255
256     /*Semaforo nao existe, retorna -1*/
257     if(index == -1)
258        return RETERR;
259
260     /*Verifica se o proccesso pode fazer a operacao*/
261     if(!authorized(index))
262        return RETERR;
263
264     if (semaforos[index].valor == -MAX_SEMVAL) {
265        m_in.m_type = EBUSY;
266        return RETERR;
267     }
268
269     if (semaforos[index].pcount == TAM_FILA) {
270        m_in.m_type = EBUSY;
271        return RETERR;
272     }
273
274
275     (semaforos[index].valor)--;
276
277      if (semaforos[index].valor < 0) {
278        int last = semaforos[index].end;
279        int proc_nr = who_p;
280
281        resultado = 0;
282        if (semaforos[index].pcount == 0) {
283           semaforos[index].fila_pid[last] = proc_nr;
284        }else if (last == (TAM_FILA - 1)) {
285           last = 0;
286           semaforos[index].end = last;
287           semaforos[index].fila_pid[last] = proc_nr;
288        }else {
289           ++last;
290           semaforos[index].end = last;
291           semaforos[index].fila_pid[last] = proc_nr;
292        }
293
294        (semaforos[index].pcount)++;
295
296
297        return (SUSPEND);
298     }
299
300     return resultado;
```

```
301 }
302
303 /*****************************************************FREE SEM
        ************************************************************/
304 PUBLIC int do_freesem (void) {
305
306     int resultado = RETERR;
307
308     int sem = m_in.m1_i1;
309     int i;
310
311
312     int index;
313
314     /*Passou valor errado*/
315     if (sem <= 0 || sem > INT_LIMIT - 1) {
316        m_in.m_type = EINVAL;
317        return RETERR;
318     }
319
320     /*Procura indice do semaforo no vetor de semaforos*/
321     for (i = 0; i < NR_SEMS; i++){
322        if(semaforos[i].id == sem){
323            index = i;
324            break;
325        }
326     }
327
328     /*Semaforo nao existe*/
329     if(index == -1)
330        return RETERR;
331
332     /*Verifica se o processo eh dono para poder libera-lo*/
333     if(semaforos[index].dono != who_p)
334        return RETERR;
335
336     if (semaforos[index].pcount != 0) {
337        m_in.m_type = EBUSY;
338        return RETERR;
339     }
340
341     resultado = 0;
342     qtd_semaforos_ativos--;
343
344     semaforos[index].id = NO_SEM;
345     semaforos[index].valor = 0;
346
347     semaforos[index].pcount = 0;
348     semaforos[index].fila_pid[0] = NADA;
349     semaforos[index].begin = 0;
350     semaforos[index].pcount = 0;
351
352     return resultado;
353 }
```