Universidade de Brasília – Campus Gama – UnB - FGA
Engenharia de Software
Professor: Fernando William Cruz
Aluno: Luan Guimarães Lacerda
Matrícula: 12/0125773

Problema simulador da sala de votação utilizando semáforos e memória compartilhada na Linguagem C em ambiente Linux

Github:
https://github.com/luanguimaraesla/semaphore_voting_room_simulator

01 de Maio de 2016

0. Descrição do Problema

Primeiramente, todos os arquivos referenciados nesse texto se encontram no repositório remoto no GitHub referenciado na capa deste documento.

O problema é uma abstração computacional de uma votação na qual participam deputados, vereadores e senadores. Para votar, cada um dos oficiais deve adentrar em uma sala na qual existem regras básicas que devem ser seguidas para manter a ordem da votação.

São cinco regras que devem ser seguidas:

I. Os senadores só podem entrar para votar se a sala estiver vazia;
II. Até 5 deputados podem entrar e votar siultaneamente na sala de votação;
III. Não há limites para a quantidade de vereadores que podem entrar na sala;
IV. Vereadores e deputados podem entrar juntos seguindo as regras anteriores;
V. Antes de entrar na sala para votar cada oficial deve permanecer um tempo pensando em seu voto.

Além disso, alguns requisitos arquiteturais foram solicitados de acordo com os objetivos do trabalho. Portanto, para concluir o projeto com êxito, o programa deveria ser feito em Linguagem C no ambiente Linux utilizando memória compartilhada e semáforos. O número de oficiais poderia ser resgatado a partir do teclado ou gerado randomicamente. Cada oficial seria representado por um processo. No decorrer do programa, cada processo deveria imprimir na tela o seu resultado de forma a deixar claro o andamento do programa.

1. Metodologia do Grupo

Para uma maior compreensão dos conteúdos trabalhados dentro desse projeto, exergando a importância da temática para um promissor futuro profissional trabalhando com desenvolvimento de softwares embarcados ou em baixo nível, o autor optou por realizar o trabalho sem a formação de um grupo, sendo unicamente responsável por todas as etapas do desenvolvimento do projeto.

Devido a simplicidade da solução nenhuma metodologia em especial foi elaborada, exceto a utilização intensiva da ferramenta Git, a qual está presente hoje na maioria dos projetos de software. Algumas ***issues*** (tarefas a serem realizadas/problemas a serem solucionados) foram criadas com o auxílio da plataforma de repositórios remotos GitHub para realizar algumas melhorias, de forma a obter um trabalho direcionado e organizado.

2. Solução

Inicialmente, devido a inexperiência do autor com o uso de semáforos compartilhados entre diferentes processos, a solução se mostrava um tanto obscura. Então, levando em consideração os problemas de concorrência discutidos no Capítulo 2 do livro Sistemas Operacionais do professor Tanenbaum, alguns algorítmos começaram a ser elaborados de forma a esclarecer a integração entre processos, memória compartilhada e semáforos. Esses programas podem ser encontrados na pasta *“learning/”* do projeto.

Após curto período de aprendizado inciou-se a construção da solução do problema. Essa é o conjunto de três módulos que atuam em conjunto para simular a votação em todos os aspéctos. São eles:

- officials.h

  Esse módulo contém as funções principais que regem o comportamento de cada oficial e do conjunto de oficiais. Será feita uma breve discussão sobre as funções principais responsáveis diretamente pelo funcionamento adequado do programa:

  - void vote(int official_id, official_type official);

    Função que recebe o identificador do oficial e seu tipo que pode ser *senator*, *alderman* ou *assemblyman*, imprimindo na saída padrão uma mensagem com as informações de tal oficial.

  - void think(int official_id, official_type official);

    Função que recebe os mesmos argumentos da explicada anteriormente e imprime as informações de tal oficial seguidas da mensagem “[THINKING]”. Também faz o processo dormir por um tempo gerado randomicamente.

  - void increment_official_counter(official_type official);

    Faz o uso de um mutex para bloquear o acesso ao contador de pessoas e de oficiais de cada tipo (motivo do parâmetro *official_type*) dentro da sala e então incrementa essas variáveis, desbloqueando a mutex após os incrementos.

  - void decrement_official_counter(official_type official);

    Realiza o mesmo procedimento que a variável anterior, porém decrementa as variáveis de controle.

  - void enter_in_the_voting_room(int official_id, official_type official);

    É a função mais importante do sistema e determina o controle de todo o processo de votação. Para facilitar a visualização faremos uma abordagem passo a passo.

    1. Primeiramente chama a função *think* descrita acima dando início ao protocolo de votação;

2. Verifica se o oficial que está querendo votar é do tipo senador. Se for, o seguinte algorítmo é executado

```
70   if(official == senator){
71     while(1){
72       // Block the voting room controller's variables access
73       sem_wait(&(vrctl->check_counter_mutex));
74       // If there is no officials into the voring room,
75       // the senator can vote, note the mutex still blocked
76       if(vrctl->official_counter == 0) break;
77       // Else, unlock the mutex and continue looping
78       else sem_post(&(vrctl->check_counter_mutex));
79     }
80     // Increment voting room controller's variables
81     (vrctl->official_counter)++;
82     (vrctl->number_of_senators)++;
83
84     vote(official_id, official); // vote
85
86     // Decrement voting room controller's variables
87     (vrctl->official_counter)--;
88     (vrctl->number_of_senators)--;
89
90     // Unlock the mutex
91     sem_post(&(vrctl->check_counter_mutex));
92   }
```

Os comentários no código são explicativos, entretanto vale destacar a forma como apenas um senador pode votar por vez e sozinho na sala. Quando este bloqueia o mutex de leitura/escrita das variáveis de controle e percebe que não há nenhum outro oficial dentro da sala, o mutex não é desbloqueado, portanto nenhum outro processo pode alterar as variáveis de controle da sala de votação, sendo assim se torna impossível para qualquer outro oficial (processo) obter acesso à sala, já que o primeiro passo seria alterar as variáveis de controle obtendo o mutex.

3. Se for um vereador, o processo é simples, apenas espera pelo mutex de acesso às variáveis de controle da sala, através da função explicada acima *increment_official_counter*, vota e então chama *decrement_official_counter*, para dizer que está saindo da sala e decrementar o contador de total de oficiais e do total de veradores. Nenhuma política em especial é atribuida a vereadores já que esses podem entrar para votar em qualquer quantidade de oficiais.

4. Se for um deputado, um *down()* é chamado para decrementar um semáforo que inicia em 5, permitindo que apenas esse número de deputados seja permitido estar presente simultaneamente na sala de votação. *increment_official_counter* é chamado para se ter acesso às variáveis de controle, o deputado vota, *decrement_official_counter* é chamado para decrementar os contadores do total de pessoas e total de deputados e antes de sair da função, um *up()* é dado no semáforo que controla o número de deputados permitidos dentro da sala, liberando o acesso para outro.

- void create_officials(int *pids, int number_of_officials, official_type type);
  Recebe um vetor de inteiros que conterá os pids dos processos filhos criados, o número de processos filhos que devem ser criados e qual o tipo de oficial esses processos simularão. A variável counter funciona como o identificador de cada oficial.

```
113   int counter = 0;
114
115   while(counter < number_of_officials){
116     if((*(pids + counter) = fork()) < 0){
117       fprintf(stderr, "Error: forking official.\n");
118     }else if(*(pids + counter) == 0){
119       enter_in_the_voting_room(counter, type);
120       exit(0);
121     }else{
122       counter++;
123       continue;
124     }
125   }
```

- voting_room.h
  Conjunto de funções/estruturas necessárias para controlar o funcionamento da sala de votações. Basicamente possui uma única função de inicialização dos parâmetros da sala de votação e a estrutura principal de controle de todo o software. Essa estrutura é acessível via memória compartilhada por todos os processos.

```
 7 struct voting_room_controller{
 8   sem_t check_counter_mutex;
 9   int official_counter;
10   sem_t assemblyman_counter;
11   int number_of_senators;
12   int number_of_assemblymen;
13   int number_of_aldermen;
14 };
```

  As variáveis inteiras servem para contar a quantidade de oficiais presentes dentro da sala de votação, *official_counter* conta o total de oficiais e os outros contadores contam respectivamente o número de senadores, deputados e vereadores.

  O semáforo *check_counter_mutex* é utilizado como uma variável de exclusão mútua para proteger o acesso aos contadores da estrutura de controle, apenas um processo pode obter o acesso a essas variáveis por vez.

  O semáforo *assemblyman_counter* é inicializado com o valor 5 e é responsável por permitir que apenas 5 deputados possam adentrar na sala de votação por vez.

- simulator.h

  Esse é o módulo operador cuja função é iniciar a memória compartilhada e inserir a estrutura de controle nela, gerar randomicamente o número de oficiais, chamar as funções de criação de processos filhos e então aguardar até que esses processos terminem para destruir os semáforos e a memória compartilhada antes de finalizar a execução do programa.

  Para auxiliar em uma criação homogênea de processos, de forma que senadores, deputados e vereadores sejam executados em ordem aleatória, três threads foram criadas para a execução das funções de criação de cada um dos tipos simultâneamente. Essa foi uma correção apontada, já que antes, quando se criava os senadores primeiro, por exemplo, todos eles votavam antes que outros processos solicitassem entrar na sala.

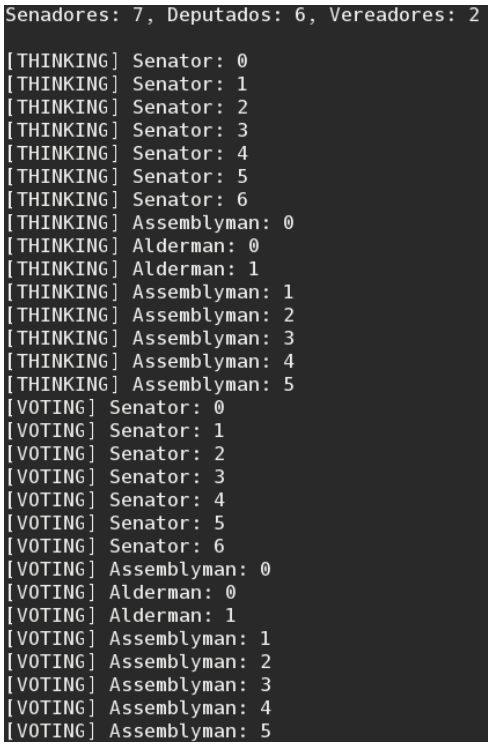
3. Opinião do autor sobre o projeto

Se comparada com as soluções de problemas clássicos de concorrência entre processos como o “Jantar dos Filósofos” observa-se uma simplicidade maior na elaboração da solução do problema da sala de votação utilizando-se apenas dois semáforos para organizar a complexa interação entre os processos.

Entretanto, o mecanismo no qual se dá o compartilhamento de semáforos entre diferentes processos se tornou claro e agora faz parte de uma lista de ferramentas das quais o autor tem conhecimento para resolver diversos problemas que antes pareciam impossíveis.

É evidente que o comportamento do programa não segue uma lógica preditiva já que não há o controle sobre o escalonador de processos. Para um cenário idêntico há uma série de fatores que influênciam a execução dos processos e interferem no resultado final. Abaixo segue uma tabela com três testes executados na primeira versão do programa.

CASO TESTE 1



CASO TESTE 2

```
Senadores: 9, Deputados: 5, Vereadores: 7

[THINKING] Senator: 0
[THINKING] Senator: 1
[THINKING] Senator: 2
[THINKING] Senator: 3
[THINKING] Senator: 4
[THINKING] Assemblyman: 0
[THINKING] Assemblyman: 1
[THINKING] Assemblyman: 2
[THINKING] Assemblyman: 3
[THINKING] Assemblyman: 4
[THINKING] Alderman: 0
[THINKING] Senator: 5
[THINKING] Senator: 6
[THINKING] Senator: 7
[THINKING] Senator: 8
[THINKING] Alderman: 1
[THINKING] Alderman: 2
[THINKING] Alderman: 3
[THINKING] Alderman: 4
[THINKING] Alderman: 5
[THINKING] Alderman: 6
[VOTING] Senator: 0
[VOTING] Senator: 1
[VOTING] Senator: 2
[VOTING] Senator: 3
[VOTING] Senator: 4
[VOTING] Assemblyman: 0
[VOTING] Assemblyman: 1
[VOTING] Assemblyman: 2
[VOTING] Assemblyman: 3
[VOTING] Assemblyman: 4
[VOTING] Alderman: 0
[VOTING] Senator: 5
[VOTING] Senator: 6
[VOTING] Senator: 7
[VOTING] Senator: 8
[VOTING] Alderman: 1
[VOTING] Alderman: 2
[VOTING] Alderman: 3
[VOTING] Alderman: 4
[VOTING] Alderman: 5
[VOTING] Alderman: 6
```

CASO TESTE 3

```
Senadores: 1, Deputados: 14, Vereadores: 2

[THINKING] Assemblyman: 0
[THINKING] Assemblyman: 1
[THINKING] Assemblyman: 2
[THINKING] Assemblyman: 3
[THINKING] Assemblyman: 4
[THINKING] Assemblyman: 5
[THINKING] Assemblyman: 6
[THINKING] Assemblyman: 7
[THINKING] Assemblyman: 8
[THINKING] Assemblyman: 9
[THINKING] Assemblyman: 10
[THINKING] Assemblyman: 11
[THINKING] Assemblyman: 12
[THINKING] Assemblyman: 13
[THINKING] Alderman: 0
[THINKING] Senator: 0
[THINKING] Alderman: 1
[VOTING] Assemblyman: 0
[VOTING] Assemblyman: 1
[VOTING] Assemblyman: 2
[VOTING] Assemblyman: 3
[VOTING] Assemblyman: 4
[VOTING] Assemblyman: 5
[VOTING] Assemblyman: 6
[VOTING] Assemblyman: 7
[VOTING] Assemblyman: 8
[VOTING] Assemblyman: 9
[VOTING] Assemblyman: 10
[VOTING] Assemblyman: 11
[VOTING] Assemblyman: 12
[VOTING] Assemblyman: 13
[VOTING] Alderman: 0
[VOTING] Senator: 0
[VOTING] Alderman: 1
```

Algumas conclusões podem ser retiradas dessas execuções. Como por exemplo, a dinâmica aumentaria bastante se o oficial pensase assim que entra na sala e não antes de tentar, já que, mesmo com valores randomicos de tempo, a ordem de chegada para tentar entrar na sala é praticamente a mesma da ordem de execução. Outro ponto de melhoria é dizer quando um oficial sai da sala com a informação [EXITING]. Por último, a dinâmica da votação seria simulada de maneira mais fiel caso fossem estipulados tempos randômicos para votar e para sair da sala.

Dados esses pontos de melhoria, a segunda versão do programa foi elaborada e é a atual versão final do projeto, contendo novas informações e adicionando uma interface mais amigável para uma interpretação mais consistente dos dados.

CASO TESTE 4

```
Senadores: 1, Deputados: 7, Vereadores: 1

[THINKING] Senator: 0
[THINKING] Assemblyman: 0
[VOTING] Senator: 0
[THINKING] Assemblyman: 1
[THINKING] Assemblyman: 2
[THINKING] Assemblyman: 3
[THINKING] Assemblyman: 4
[THINKING] Assemblyman: 5
[THINKING] Assemblyman: 6
[THINKING] Alderman: 0
[EXITING] Senator: 0
[VOTING] Assemblyman: 0
[VOTING] Assemblyman: 1
[VOTING] Assemblyman: 4
[VOTING] Alderman: 0
[VOTING] Assemblyman: 3
[VOTING] Assemblyman: 2
[EXITING] Assemblyman: 0
[EXITING] Assemblyman: 1
[EXITING] Alderman: 0
[EXITING] Assemblyman: 3
[EXITING] Assemblyman: 4
[VOTING] Assemblyman: 5
[VOTING] Assemblyman: 6
[EXITING] Assemblyman: 2
[EXITING] Assemblyman: 5
[EXITING] Assemblyman: 6
```

CASO TESTE 5

```
Senadores: 3, Deputados: 3, Vereadores: 5

[THINKING] Assemblyman: 0
[THINKING] Assemblyman: 1
[THINKING] Assemblyman: 2
[VOTING] Assemblyman: 0
[VOTING] Assemblyman: 1
[THINKING] Senator: 0
[VOTING] Assemblyman: 2
[THINKING] Senator: 1
[THINKING] Senator: 2
[THINKING] Alderman: 0
[EXITING] Assemblyman: 0
[EXITING] Assemblyman: 1
[THINKING] Alderman: 1
[VOTING] Alderman: 0
[EXITING] Assemblyman: 2
[THINKING] Alderman: 3
[VOTING] Alderman: 1
[VOTING] Alderman: 3
[THINKING] Alderman: 4
[EXITING] Alderman: 0
[THINKING] Alderman: 2
[VOTING] Alderman: 4
[EXITING] Alderman: 1
[VOTING] Alderman: 2
[EXITING] Alderman: 3
[EXITING] Alderman: 4
[EXITING] Alderman: 2
[VOTING] Senator: 2
[EXITING] Senator: 2
[VOTING] Senator: 1
[EXITING] Senator: 1
[VOTING] Senator: 0
[EXITING] Senator: 0
```

Dessa maneira é fácil notar como a interação dos processos foi beneficiada e deixou a simulação mais próxima da realidade, que é o objetivo. Alguns pontos de melhorias são evidentes como a exibição dos contadores dos presentes na sala em tempo real, de forma que seja possível monitorar com facilidade a dinâmica do fluxo dos oficiais adentrando e saindo em sequência.

4. Descrição da utilização de semáforos não binários e mutexes

4.1. Semáforos não binários

Semáforos são uma excelentes ferramentas quando se busca obter o controle da multiprogramação. É a partir deles que se é possível sincronizar e estabelecer regras de concorrências entre processos, possibilitando uma série de abordagens, principalmente se tratando de simulações nas quais diversas atividades são executadas paralelamente.

As funções básicas de semáforos não binários giram em torno da criação, utilização e destruição do mesmo. Na criação, usa-se a função *int sem_init(sem_t *sem, int pshared, unsigned int value);* da biblioteca semaphore.h. Essa função recebe respectivamente os parâmetros variável semáforo, inteiro 1 para compartilhamento entre processos e 0 para compartilhamento entre threads e um inteiro não sinalizado para um valor positivo de inicialização do semáforo.

A utilização comum é dada pelas funções *sem_post* e sem_wait, respectivamente *up()* e *down()*. E por último, a destruição do mesmo com a chamada *int sem_destroy(sem_t *sem).*

4.2. Mutexes

Mutexes são semáforos binários que servem para, como o próprio nome sugere, possibilitar a exclusão mútua, isto é, proteger regiões críticas do código. Podem ser uma abstração de um semáforo que inicia em ou ou utilizar a biblioteca *pthread.h* para implementar a versão POSIX dos mutexes.

Alguns exemplos de semáforos e mutexes são mostrados no problema implementado a seguir, o jantar dos filósofos que pode ser encontrado no diretório *"learning/"*.

```
 6 #define N 5
 7 #define LEFT (i+N-1)%N
 8 #define RIGHT (i+1)%N
 9 #define THINKING 0
10 #define HUNGRY 1
11 #define EATING 2
12
13 int state[N];
14 pthread_mutex_t mutex;
15 sem_t s[N];
16 pthread_t fil[N];
17
18 void test(int i){
19   if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING){
20     state[i] = EATING;
21     sem_post(&(s[i]));
22   }
23 }
24
25 void think(int i){
26   printf("%d: THINKING\n", i);
27 }
28
29 void eat(int i){
30   pthread_mutex_lock(&mutex);
31   if(state[i] == EATING)
32     printf("%d: EATING\n", i);
33   pthread_mutex_unlock(&mutex);
34 }
35
36 void take_forks(int i){
37   pthread_mutex_lock(&mutex);
38   state[i] = HUNGRY;
39   test(i);
40   pthread_mutex_unlock(&mutex);
41   sem_wait(&(s[i]));
42 }
43
44 void put_forks(int i){
45   pthread_mutex_lock(&mutex);
46   state[i] = THINKING;
47   test(LEFT);
48   test(RIGHT);
49   pthread_mutex_unlock(&mutex);
50 }
51
```

```
51
52 void * filosofo(void * i_aux){
53   int i = (int) i_aux;
54
55   while(1){
56     think(i);
57     sleep(1);
58     take_forks(i);
59     eat(i);
60     put_forks(i);
61   }
62
63   pthread_exit(NULL);
64 }
65
66 int main(){
67   pthread_mutex_init(&mutex, NULL);
68   int a;
69   for(a = 0; a < N; a++)
70     sem_init(&(s[a]), 0, 0);
71
72   for(a = 0; a < N; a++)
73     pthread_create(&fil[a], NULL, filosofo, (void *) a);
74
75   for(a = 0; a < N; a++)
76     pthread_join(fil[a], NULL);
77
78
79   for(a = 0; a < N; a++)
80     sem_destroy(&(s[a]));
81   pthread_mutex_destroy(&mutex);
82
83 }
```

5. Referências

[Tanenbaum, 2003] Tanenbaum, A. Sistemas Operacionais Modernos. 2a. Ed