

Universidade Federal de Roraima

Rua, Av. Cap. Ene Garcês, 2413 - Aeroporto,
Boa Vista - RR, 69310-000

Aluno: Tarlison Sander Lima Brito
Matrícula: 2017013008

Laboratório de Sistemas Operacionais

23 de maio de 2019

[Questão 1]: (Prática - A)

The screenshot displays the SOsim operating system simulator interface. The main window is titled "Console SOsim" and contains several sub-windows:

- Gerência de Processos (Process Management):** A table showing the state of processes. The table has columns: Cor, PID, Prio, Estado, Temp UCP, and Frames. Two processes are listed: PID 5252315 (green) and PID 5252421 (red), both in the "Pronto" state with a priority of 0 and a time slice of 20. Buttons for "Criar", "Suspender", "Prosseguir", "Finalizar", "Prioridade", and "PCB" are visible.
- Gerência de Memória (Memory Management):** A table showing memory allocation. The table has columns: Cor, PID, and a grid of memory slots. The first 10 slots are filled with green and red dots, indicating memory usage. Buttons for "Tam. LPL" and "Tam. LPM" are visible.
- Gerência do Processador (Processor Management):** A section showing the execution of processes. It includes a "Pronto" queue with a vertical bar chart and a "Execução" section with a horizontal bar chart. Buttons for "Espera", "I/O", and "Susp" are visible.
- Estatísticas (Statistics):** A section showing various system statistics. The statistics include: Hora do Início (13:52:44), Tempo Decorrido (91), Número de Processos (2), Processos Escalonados (40), Tempo Total de UCP (s) (40), Estado Pronto Espera (s) (36.400,00), Memória (bytes) (300), Total Page Fault (10), and others.

Porque os processos estão sendo escalonados através de escalonamento circular e assim eles possuem uma ordem para serem executados. E esse tipo de escalonamento é organizado de uma maneira que cada um deles possua um determinado tempo da CPU e caso um desses processos não termine dentro do seu tempo ele é colocado no fim da fila e outro tempo é dado para o processo no começo da fila.

Com isso podemos ver na imagem em que os dois processos estão prontos e apenas esperando serem escalonados que o próximo processo será o verde, observável a partir da aba gerência de processos, pois é o tempo e a vez dele de ser executado e ele apenas está esperando o delay que ocorre até ele começar executar.

(Prática - B)

The screenshot displays the SOsim console interface, which is divided into several panels:

- Console SOsim:** Shows system status including "Tempo 284", "Processos 2", and "Memória Livre 90%". It includes a "Parar" button.
- Gerência de Processos:** A table showing the state of processes. The table has columns: Cor, PID, Prio, Estado, Temp UCP, and Frames.

Cor	PID	Prio	Estado	Temp UCP	Frames
Verde	509802	4	Execução	124	5
Vermelho	533265	3	Pronto	0	5

 To the right of the table are buttons: Criar, Suspendir, Prosseguir, Finalizar, Prioridade, and PCB.
- Gerência de Memória:** A grid showing memory allocation across 100 frames (0-99). The first 10 frames (0-9) are marked with green dots, indicating they are in use. Below the grid are sliders for "Tam. LPL" (set to 90) and "Tam. LPM" (set to 0).
- Gerência do Processador:** Shows the execution state. A green dot indicates the current process is in execution. A vertical bar on the right shows the ready queue (Pronto) with 15 slots, where the bottom slot (0) is red, indicating a process is ready to be scheduled. On the left, there are sections for "Espera" (Waiting) with I/O and Suspended states, and sliders for "Tempo de espera de I/O", "Fatia de tempo", and "Clock da UCP".
- Estatísticas:** A summary of system performance metrics.

Hora do Início		Tempo Decorrido	
17:04:36		284	
Número de Processos	2	Prontos	2
Processos Escalonados	123	Exec	0
Tempo Total de UCP (s)	123	Espera	0
Estado Pronto Espera (s)	19.267.200,00	Throughput (Proc/s)	0,43
Memória (bytes)	800	Turnaround	
Total Page Fault	10	Utilização UCP (%)	40,00
		Média (s)	111,50
		Espera Acumulada (s)	1,00
		Memória em uso	80
		Livre	90%
		Page Faults/seg	0,04
		Hw page fault	10
		Sw page fault	0

O starvation ocorre pois os processos possuem prioridades pré-determinadas e quando o escalonador vai setar o processo ele escolhe o que possui maior prioridade e assim o processo de prioridade 4 sempre irá executar na frente do processo de prioridade 3 fazendo com que esse processo nunca seja executado.

Uma solução seria a criação de uma fila que fosse feita a partir dos pedidos dos processos para alocar os recursos e assim sempre que um processo desejar um recurso o pedido é colocado no final da fila associada a ela e assim que o recurso for liberado o sistema seleciona o primeiro processo da fila. Ou seja, esquema FIFO.

Outra solução seria diminuir a prioridade do processo que está tomando a CPU ou até mesmo excluí-lo se possível para que os outros processos possam ser executados.

(Prática - C)

The screenshot displays the SOsim simulator interface with several active windows:

- Console SOsim:** Shows system status: Tempo 789, Processos 2, Memória Livre 90%, and a Parar button.
- Gerência de Processos:** A table listing processes:

Cor	PID	Prio	Estado	Temp UCP	Frames
Green	2130854	0	Execução	169	5
Red	2130958	0	Pronto	163	5

 Buttons: Criar, Suspendir, Prosseguir, Finalizar, Prioridade, PCB.
- Gerência de Memória:** A 10x10 grid representing memory frames. The first column (index 0) contains green circles, and the second column (index 1) contains red circles. Below the grid are sliders for Tam. LPL (set to 90) and Tam. LPM (set to 0).
- Contexto do Processo:** Shows PCB and Tab. Pág. for PID 2130958. A table of page frames:

NPV	NPR	Bit V	Bit M	Local
0	1	1	1	MP
1	3	1	0	MP
2	5	1	0	MP
3	7	1	0	MP
4	9	1	0	MP
- Gerência do Processador:** Shows a bar chart for 'Pronto' (blue bars) and 'Execução' (green bar). It also includes 'Espera' (I/O and Susp) and a 'Clock da UCP' section with sliders for 'Tempo de espera de I/O', 'Fatia de tempo', and 'Clock da UCP'.
- Estatísticas:** A summary of system performance:

Hora do Início	16:19:56	Tempo Decorrido	789
Número de Processos	2	Prontos	2
Processos Escalonados	341	Exec	0
Tempo Total de UCP (s)	331	Espera	0
Estado Pronto Espera (s)	172.800,00	Throughput (Proc/s)	0,43
Memória (bytes)	800	Utilização UCP (%)	40,00
Total Page Fault	10	Media Média (s)	1,00
		Espera Acumulada (s)	2,72
		Memória em uso	80
		Livre	90%
		Page Faults/seg	0,01
		Hw page fault	10
		Sw page fault	0

O espaço de endereçamento real máximo do processo será a quantidade de espaço da memória principal e virtual juntas. Já o espaço de endereçamento real mínimo do processo será o tamanho mínimo da tabela de mapeamento carregada.

O tamanho da página virtual irá variar de acordo com o processador utilizado e também da arquitetura de hardware, em algumas arquiteturas é possível que esse tamanho seja configurado.

(Prática - D)

ades Wine ▾ 20 de maio

Console SOsim

Opções Janelas Ajuda

Tempo 1044 Processos 7 Memória Livre 69% Parar

Gerência de Processos

Cor	PID	Prio	Estado	Temp UCP	Frames
Verde	2043154	0	Pronto	127	5
Vermelho	2043257	0	Pronto	114	5
Amarelo	2104066	0	Suspenso	3	5
Púrpura	2104168	0	I/O	34	5
Azul	2104271	0	Pronto	33	5
Ciano	2320585	0	Pronto	86	5
Preto	2320699	0	Pronto	81	5

Criar Suspendir Prosseguir Finalizar Prioridade PCB

Gerência do Processador

Execução

Pronto

Espera

I/O

Susp

Tempo de espera de I/O

Fatias de tempo

Clock da UCP

Gerência de Memória

Arquivo de Paginação

Estatísticas

Hora do Início 22:20:02 Tempo Decorrido 1044

Número de Processos 7 Prontos 4 Exec 1 Espera 2

Processos Escalonados 510 Throughput (Proc/s) 0,49 Turnaround

Tempo Total de UCP (s) 478 Utilização UCP (%) 60,00

Estado Pronto Espera (s) 1.209.600,00 Média (s) 3,50 Espera Acumulada (s) 6,26

Memória (bytes) 800 Memória em uso 248 Livre 69%

Total Page Fault 31 Page Faults/seg 0,03 Hw page fault 31 Sw page fault 0

É analisado qual dos processos que serão alocados que não possuem uma previsão de utilizar a CPU nos próximos instantes e assim ele manda esse processo para o arquivo de paginação.

Para que o processo seja transferido de volta para a memória principal é usado a política por demanda a qual uma página somente será carregada quando for referenciada assim ele irá levar para a memória principal somente as que forem realmente necessárias.

[Questão 2]:

O algoritmo funciona para solucionar um impasse explicado em que um banqueiro de uma pequena cidade pode negociar com um grupo de clientes para os quais ele libera linhas de crédito. E esse algoritmo verifica se a liberação de uma requisição é capaz de levar a um estado inseguro.

É possível verificar se um estado é seguro ou não observando esta tabela abaixo(único recurso). Estão disponíveis 7 créditos para serem usados pelos clientes e o banqueiro precisa gerenciar isso.

Esse estado mostrado na tabela é seguro, pois, com essa quantidade de recursos disponíveis ele pode dar a um dos clientes a quantidade que ele precisa para atingir o crédito máximo e assim que ele terminar de usar os créditos eles são devolvidos e assim os outros podem usá-lo e desta forma todos conseguem obter o crédito.

Caso o crédito disponível fosse, por exemplo, 3 não seria possível ele administrar isso para que todos pudessem usar, considerado assim um caso inseguro. E por fim caso a requisição do cliente seja dada como insegura ela será negada, caso contrário o recurso será liberado.

Cliente A	Quantidade de crédito	Crédito máximo
A	0	6
B	0	4
C	0	5

Disponível: 7

Já quando se trata de múltiplos recursos temos duas matrizes para analisar e obviamente vários recursos para administrar.

A	3	0	1	1	X	A	1	1	0	0
B	0	1	0	0	X	B	0	1	1	2
C	1	1	1	0	X	C	3	1	0	0
D	1	1	0	1	X	D	0	0	1	0
E	0	0	0	0	X	E	2	1	1	0

$E = [6,3,4,2]$, $P = [5,3,2,2]$, $A = [1,0,2,0]$

Observando a tabela acima, a tabela do lado esquerdo indica os recursos alocados e a tabela do lado direito os recursos ainda necessários para que cada cliente atinja o máximo. O vetor E indica os recursos alocados, P os recursos disponíveis e A os recursos disponíveis que é a diferença entre o que o banqueiro tem e o que está sendo usado pelos clientes.

Considerando que o processo da linha escolhida requisita todos os recursos de que precisa e termina, é marcado como terminado e é adicionado ao vetor A todos os recursos que lhe pertenciam. Caso nenhum das linhas da tabela ao lado direito possua uma necessidade de recursos inferiores ou iguais a A, resultará em uma situação de impasse já que nenhum dos clientes poderá obter todos os créditos. E esses passos se repetem até que todos os clientes sejam atendidos ou até que haja um impasse.

```

~/Documentos/UFRR/4 Semestre/Sistemas Operac
File Edit Selection Find View Goto Tools Project Preferences Help
banker_algorithm.cpp x
1 // Banker's Algorithm
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6     // P0, P1, P2, P3, P4 are the Process names
7     int processes, resources, i, j, k;
8     processes = 5; // Number of processes
9     resources = 4; // Number of resources
10    int alloc[5][4] = { { 3, 0, 1, 1}, //P0
11                        { 0, 1, 0, 0}, //P1
12                        { 1, 1, 1, 0}, //P2
13                        { 1, 1, 0, 1}, //P3
14                        { 0, 0, 0, 0} }; //P4
15
16    int max[5][4] = { { 4, 1, 1, 1}, //P0 |
17                     { 0, 2, 1, 2}, //P1
18                     { 4, 2, 1, 0}, //P2
19                     { 1, 1, 1, 1}, //P3
20                     { 2, 1, 1, 0} }; //P4
21
22    int avail[4] = { 0, 0, 0, 0 }; // Available Resources
23
24    int f[processes], ans[processes], ind = 0;
25

```

Aqui podemos ver os vetores utilizados no algoritmo do banqueiro para múltiplos recursos. O vetor alloc representa os recursos que estão sendo usados pelo processos (que são representados por P0, P1, P2, P3, P4), o vetor max os recursos máximos de cada processo e o vetor avail os recursos ainda disponíveis para serem distribuídos entre os processos.

Nesse caso a nossa saída será "Redistribution was not possible" pois a quantidade de recursos disponíveis é 0 (vetor avail). Imagem da saída:

```

tarlison@tarlison: ~/Documentos/UFRR/4 Semestre/Sistemas Operacionais/Tarlison_labos_rr_2019$ g++ banker_algorithm.cpp -o out
tarlison@tarlison:~/Documentos/UFRR/4 Semestre/Sistemas Operacionais/Tarlison_labos_rr_2019$ ./out
Redistribution was not possible
tarlison@tarlison:~/Documentos/UFRR/4 Semestre/Sistemas Operacionais/Tarlison_labos_rr_2019$

```

Agora no caso da imagem abaixo nós temos recursos disponíveis e a matriz de recursos alocados iguais ao exemplo dado no livro Sistemas operacionais modernos, 3ª edição do Andrew S. Tanenbaum página 281. Com esses recursos agora será possível fazer a distribuição de recursos entre todos sem que ocorra erros. Trecho do código modificado:

```

~/Documentos/UFRR/4 Semestre/Sistemas Operacionais/Tarlison_labos_rr_2019/banker_algorithm.cpp - Sublime Text (U
File Edit Selection Find View Goto Tools Project Preferences Help
banker_algorithm.cpp x
1 // Banker's Algorithm
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6     // P0, P1, P2, P3, P4 are the Process names
7     int processes, resources, i, j, k;
8     processes = 5; // Number of processes
9     resources = 4; // Number of resources
10    int alloc[5][4] = { { 3, 0, 1, 1}, //P0
11                        { 0, 1, 0, 0}, //P1
12                        { 1, 1, 1, 0}, //P2
13                        { 1, 1, 0, 1}, //P3
14                        { 0, 0, 0, 0} }; //P4
15
16    int max[5][4] = { { 4, 1, 1, 1}, //P0
17                     { 0, 2, 1, 2}, //P1
18                     { 4, 2, 1, 0}, //P2
19                     { 1, 1, 1, 1}, //P3
20                     { 2, 1, 1, 0} }; //P4
21
22    int avail[4] = { 1, 0, 2, 0 }; // Available Resources
23
24    int f[processes], ans[processes], ind = 0;
25

```


Ir  gerar esta sa da que significa a ordem que ser  preciso fazer a distribui  o para que n o ocorra erros (no caso P3 -> P4 -> P0 -> P1 -> P2):

```

tarlison@tarlison: ~/Documentos/UFRR/4 Semestre/Sistemas Operacionais/Tarlison_labos_rr_2019
tarlison@tarlison:~/Documentos/UFRR/4 Semestre/Sistemas Operacionais/Tarlison_labos_rr_2019$ g++ banker_algorithm.cpp -o out
tarlison@tarlison:~/Documentos/UFRR/4 Semestre/Sistemas Operacionais/Tarlison_labos_rr_2019$ ./out
The safe redistribution sequence is:
P3 -> P4 -> P0 -> P1 -> P2
tarlison@tarlison:~/Documentos/UFRR/4 Semestre/Sistemas Operacionais/Tarlison_labos_rr_2019$

```

No algoritmo de apenas um recurso n s temos isso bem mais simplificado. Temos o vetor alloc que   um vetor unidimensional que armazena, respectivamente, os valores alocados por P0,P1,P2,P3, o vetor max que representa os valores m ximos que cada processo aloca e a vari vel avail que possui a quantidade de recursos dispon veis.

```

~/Documentos/UFRR/4 Semestre/Sistemas Operacionais/Tarlison_labos_rr_2019/banker_algorithm_one_resource.cpp - Sublime
File Edit Selection Find View Goto Tools Project Preferences Help
banker_algorithm.cpp x banker_algorithm_one_resource.cpp x
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     // P0, P1, P2, P3 are the Process names
6     int processes, i, j, k, y = 0;
7     processes = 4;
8
9     int alloc[4] = {    1,      //P0
10                      1,      //P1
11                      2,      //P2
12                      4    }; //P3
13
14     int max[4] = {      6,      //P0
15                      5,      //P1
16                      4,      //P2
17                      7    }; //P3
18
19     int avail = 2; // Available Resources
20
21     int f[processes], ans[processes], ind = 0;
22
23     for (k = 0; k < processes; k++) {
24

```

O algoritmo funciona da mesma forma que o de múltiplos recursos, mudando apenas a dimensão de algumas variáveis, mas a ideia permanece a mesma. Na sua execução essa será a saída:

```
Q tarlison@tarlison: ~/Documentos/UFRR/4 Semestre/Sistemas Operacionais/Tarlison_labos_rr_2019
tarlison@tarlison:~/Documentos/UFRR/4 Semestre/Sistemas Operacionais/Tarlison_labos_rr_2019$ g++ banker_algorithm_one_resource.cpp -o out
tarlison@tarlison:~/Documentos/UFRR/4 Semestre/Sistemas Operacionais/Tarlison_labos_rr_2019$ ./out
The safe redistribution sequence is:
P2 -> P3 -> P0 -> P1
tarlison@tarlison:~/Documentos/UFRR/4 Semestre/Sistemas Operacionais/Tarlison_labos_rr_2019$ |
```

Mudando a quantidade de recursos para 1 já não será mais possível distribuir de uma forma que todos os processos possam usar:

```
~/Documentos/UFRR/4 Semestre/Sistemas Operacionais/Tarlison_labos_rr_2019/banker_algorithm_one_resource.cpp - Sublime
File Edit Selection Find View Goto Tools Project Preferences Help
banker_algorithm.cpp x banker_algorithm_one_resource.cpp x
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     // P0, P1, P2, P3 are the Process names
6     int processes, i, j, k, y = 0;
7     processes = 4;
8
9     int alloc[4] = {    1,      //P0
10                      1,      //P1
11                      2,      //P2
12                      4   }; //P3
13
14     int max[4] = {      6,      //P0
15                      5,      //P1
16                      4,      //P2
17                      7   }; //P3
18
19     int avail = 1; // Available Resources
20
21     int f[processes], ans[processes], ind = 0;
22
23     for (k = 0; k < processes; k++) {
24         f[k] = 0;
```

A saída é como o esperado, “Redistribution was not possible” pois não há recursos suficientes para isso.

```
Q tarlison@tarlison: ~/Documentos/UFRR/4 Semestre/Sistemas Operacionais/Tarlison_labos_rr_2019
tarlison@tarlison:~/Documentos/UFRR/4 Semestre/Sistemas Operacionais/Tarlison_labos_rr_2019$ g++ banker_algorithm_one_resource.cpp -o out
tarlison@tarlison:~/Documentos/UFRR/4 Semestre/Sistemas Operacionais/Tarlison_labos_rr_2019$ ./out
Redistribution was not possible
tarlison@tarlison:~/Documentos/UFRR/4 Semestre/Sistemas Operacionais/Tarlison_labos_rr_2019$ |
```

[Questão 3]:

O algoritmo do Barbeiro dorminhoco funciona para resolver problemas de comunicações entre processos. O problema consiste em que em uma barbearia há um barbeiro, uma cadeira de barbeiro e N cadeiras para clientes poderem esperar a sua vez.

Quando não há clientes, o barbeiro senta-se na cadeira de barbeiro e cai no sono e assim que um cliente chega ele precisa acordar o barbeiro para atendê-lo. Caso outros clientes cheguem enquanto o barbeiro estiver atendendo outro cliente eles aguardam nas cadeiras caso haja cadeiras vazias, se não apenas vão embora. O problema é programar o barbeiro e os clientes para que não ocorra situações de disputa.

O código usa semáforos para controlar as regiões críticas. Quando começa o barbeiro chega para trabalhar e logo executa a função barber que irá fazer ele bloquear os semáforos customers e ele irá verificar que o número de clientes é 0 então irá dormir.

Quando o cliente chega ele inicia também o semáforo customers e o mutex, então acorda o barbeiro caso ele não esteja atendendo ninguém, caso ele esteja o cliente irá verificar se o número de clientes é menor do que o número de cadeiras disponíveis.

Se for ele irá sentar e aguardar, acrescentando +1 ao valor da variável waiting (que é a quantidade de clientes esperando) e também dá um up no customers para acordar o barbeiro na sua vez . Se não ele sairá da barbearia e irá liberar o mutex. COmo podemos observar no código abaixo:

```
~/Documentos/UFRR/4 Semestre/Sistemas Operacionais/Tarlison_labos_rr_2019/barber_algorithm.cpp - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
barber_algorithm.cpp
1 #include <iostream>
2 #include <unistd.h>
3 #include <pthread.h>
4 #include <semaphore.h>
5 using namespace std;
6
7 #define CHAIRS 5          /* number of seats for customers waiting. */
8 #define TRUE 1
9
10 sem_t customers;          /* customer number waiting for service.*/
11 sem_t barbers;           /* number of barbers waiting for customers.*/
12 sem_t mutex;             /* for mutual exclusion. */
13 int waiting = 0;         /* customers who are waiting. */
14
15 void cut_hair() {
16     printf("The barber is cutting the client's hair\n");
17     sleep(3);
18 }
19
20 void *barber(void *arg) {
21     while(TRUE) {
22         sem_wait(&customers); /* go to sleep if the number of clients is 0.*/
23         sem_wait(&mutex);     /* gets access to 'waiting'.*/
24         waiting = waiting - 1; /* decreases from a waiting client count because it will be serviced.*/
25         sem_post(&barbers);   /* a barber is now ready to cut hair.*/
26         sem_post(&mutex);     /* free 'waiting'. */
27         cut_hair();           /* cuts hair. */
28     }
29
30     pthread_exit(NULL);
31 }
32
```



```
~/Documentos/UFRR/4 Semestre/Sistemas Operacionais/Tarlison_labos_rr_2019/barber_algorithm.cpp - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help

barber_algorithm.cpp x
31 }
32
33 void *customer(void *arg) {
34     sem_wait(&mutex);
35     /*If the amount of waiting customers is less than
36     the amount of chairs means there is room to sit and wait.*/
37     if(waiting < CHAIRS){
38         cout << "Customer arrived to cut hair" << endl;
39         waiting = waiting + 1;
40         sem_post(&customers);
41         sem_post(&mutex);
42         sem_wait(&barbers);
43         cout << "The customer is having his hair cut" << endl;
44     }
45
46     else{ /*The barber is full and customer will not wait.*/
47         sem_post(&mutex);
48         cout << "The customer gave up (the hall is very full)" << endl;
49     }
50
51     pthread_exit(NULL);
52 }
53
54
55
56
57 int main() {
58     sem_init(&customers, TRUE, 0);
59     sem_init(&barbers, TRUE, 0);
60     sem_init(&mutex, TRUE, 1);
61
62     pthread_t b, c;
```

Dependendo do número de cadeiras e do escalonamento das threads as cadeiras ficarão cheias mais rápido ou não. Essa será a saída considerando 5 cadeiras, se for alterado o código o mesmo acontecerá quando apenas mudará a quantidade de clientes que poderão ficar esperando:

[illegible]

Códigos e como compilar os mesmos estão disponíveis no github:

https://github.com/Tarlison/Tarlison_labos_rr_2019.git

Este arquivo será disponibilizado em .odt, .docx e .pdf para evitar problemas de compatibilidade e/ou desformatação do mesmo.