Sistemas Operacionais
Terceira Lista de Exercícios – Solução

Norton Trevisan Roman

14 de novembro de 2013

5. Monothread:

   em média, o tempo gasto a cada 3 operações será 15ms + 15ms + (15+75)ms = 120ms. Com 3 requisições a cada 120ms, temos 25 requisições/s.

   Multithread:

   Com cada requisição rodando em uma thread separada, as threads que bloqueiam não atrapalham as demais. Descontando-se o tempo de escalonamento, e supondo um fluxo constante de pedidos, teremos um pedido a cada 15ms (como se nenhuma bloqueasse), ou 66requisições/s.

7. Sim. Após cada chamada a pthread_create execute pthread_join(threads[i],null), para que o programa principal espere a thread recém criada terminar, para então prosseguir no laço que cria uma nova thread.

10. Cada thread chama seus próprios procedimentos, então deve ter sua própria pilha para as variáveis locais, parâmetros e endereços de retorno. Isso é verdade tanto para threads no nível do usuário quanto para threads de núcleo.

12. Certamente, se o computador aceitar multiprogramação. Por exemplo, um processo A pode ser alguma variável compartilhada. Então a simulação do clock o interrompe e roda o processo B. Ele também lê a mesma variável, incrementando-a. Quando a nova simulação do clock acontecer, botando A para rodar, ele estará com o valor desatualizado da variável.

15. Sim. Pode-se bloquear o acesso à memória via TSL.

16. No caso preemptivo, sim. No caso não preemptivo, pode falhar. Considere, por exemplo, quando turn é inicialmente 0, ambos os processos foram marcados como interessados, e o processo 1 roda primeiro. Teremos um laço infinito, pois ele nunca deixará a CPU.

17. Com threads do núcleo, uma thread pode bloquear num semáforo e o núcleo pode rodar outra thread do mesmo processo – não há problema. Já no espaço do usuário, quando a thread bloquear, o núcleo achará que o processo bloqueou. O problema é que somente outra thread (do mesmo processo) desbloqueará o processo, thread esta que nunca será rodada, pois o processo está bloqueado – falha.

18. A comunicação é feita passando-se mensagens. No unix: pipes.

19. Esse esquema é basicamente espera ocupada – uma das soluções para evitar-se condições de corrida. Assim, não leva a elas, pois nada se perde nunca.

21. Pelo uso de uma variável compartilhada.

25. (a) Tira-se, ao todo, 200+100=300 de A, e deposita-se, ao todo 100+200=300 em B. Logo, A = 200 e B = 1200.

    (b) Considere a sequência de operações abaixo:

Op.	x	y	A	B
–	–	–	500	900
1a	500	–	500	900
2a	500	500	500	900
1b	300	500	500	900
2b	300	400	500	900
1c	300	400	300	900
2c	300	400	400	900
1d	900	400	400	900
2d	900	900	400	900
1e	1000	900	400	900
2e	1000	1100	400	900
1f	1000	1100	400	1000
2f	1000	1100	400	1100

Então A = 400 e B = 1100

(c) Deve-se proteger as sequências de leitura + modificação + atualização, para cada cliente. Ou seja:

```
Processo 0 (Define os semáforos compartilhados)
int S1 = 1;
int S2 = 1;

Processo 1 (Cliente A)
/* saque em A */
    down(S1);
1a. x := saldo_do_cliente_A;
1b. x := x - 200;
1c. saldo_do_cliente_A := x;
    up(s1);

/* deposito em B */
    down(S2);
1d. x := saldo_do_cliente_B;
1e. x := x + 100;
1f. saldo_do_cliente_B := x;
    up(s2);
```

```
Processo 2 (Cliente B)
/* saque em A */
    down(S1);
2a. y := saldo_do_cliente_A;
2b. y := y - 100;
2c. saldo_do_cliente_A := y;
    up(s1);

/* deposito em B */
    down(S2);
2d. y := saldo_do_cliente_B;
2e. y := y + 200;
2f. saldo_do_cliente_B := y;
    up(s2);
```

26.
```
#define N 100 /* número de posições do buffer*/
typedef int semaphore;
semaphore mutex = 1; /*controla o acesso a RC*/
semaphore empty = N; /*conta as posições vazias do buffer*/
semaphore full = 0; /*conta as posições ocupadas do buffer*/
void producer (void)
{
   int item;
   while (1){
    item = produce_item( ); /*produz um novo item*/
    __down(&empty);_______
    __down(&mutex);_______
    enter_item(item); /*coloca novo item no buffer*/
    __up(&mutex);_________
    __up(&full);__________
    }
}

void consumer(void)
{
int item;
```

```
while (1){
   __down(&full);__________
   __down(&mutex);_________
   item = remove_item( ); /*retira 1 item do buffer*/
   __up(&mutex);_________
   __up(&empty);_________
   consume_item(item); /*consome um novo item no buffer*/
   }
}
```