

LISTA DE EXERCÍCIOS PARA P2 – INCLUI PROVAS ATÉ 2007/2

GRAFO DE PRECEDÊNCIA

1) Desenhe o grafo de precedência do código abaixo:

```
main()
{
[1]   int f1, f2, f3; /* Identifica processos filho*/

[2]   printf("Alo do pai\n");
[3]   f1 = fork( codigo_do_filho ); /* Cria filho 1 */
[4]   printf("Filho 1 criado\n");
[5]   f2 = fork( codigo_do_filho ); /* Cria filho 2 */
[6]   printf("Filho 2 criado\n");
[7]   wait( f1);
[8]   printf("Filho 1 morreu\n");
[9]   f3 = fork( codigo_do_filho ); /* Cria filho 3 */
[10]  printf("Filho 3 criado\n");
[11]  wait( f3);
[12]  printf("Filho 3 morreu\n");
[13]  wait( f2);
[14]  printf("Filho 2 morreu\n");
[15]  exit();
}

codigo_do_filho()
{
[16]  printf("Alo do filho\n");
[17]  exit();
}
```

SEMÁFOROS

2) O problema dos leitores/escritores consiste de um texto que pode ser lido ou escrito por vários processos. Considerando o código abaixo, responda justificando:

- É possível vários leitores lerem ao mesmo tempo ?
- É possível vários escritores escreverem ao mesmo tempo ?
- É possível postergação indefinida de um leitor ?
- É possível postergação indefinida de um escritor ?

```
int nl = 0;
semaphore tipo = 1;
semaphore exclusivo = 1;
```

<pre>void leitor(void) { ... P(exclusivo); if(nl > 0) ++ nl; else{ P(tipo); nl = 1; } V(exclusivo); Acessa o texto P(exclusivo); -- nl; if(nl == 0) V(tipo); V(exclusivo); ... }</pre>	<pre>void escritor(void) { ... P(tipo); Acessa o texto V(tipo); ... }</pre>
--	--

3) Considerando o código abaixo, uma variação da implementação do produtor/consumidor vista em aula, responda as seguintes perguntas (justificando):

- Está garantido o acesso exclusivo à variável "buffer" no caso de 1 produtor e 1 consumidor ?
- É possível a ocorrência de deadlock no caso de 2 produtores e 2 consumidores ?

```
struct tipo_dado buffer;
semaphore espera_vaga = 1;
semaphore espera_dado = 0;
```

<pre>void produtor(void) { ... P(espera_vaga); buffer = dado_produzido; V(espera_dado); ... }</pre>	<pre>void consumidor(void) { ... P(espera_dado); dado_a_consumir = buffer; V(espera_vaga); ... }</pre>
--	---

4) Considere o problema dos filósofos jantadores, onde 5 filósofos estão sentados em torno de uma mesa circular. A frente de cada um existe um prato e entre cada 2 filósofos existe 1 garfo. Cada filósofo vive um eterno ciclo de “pensa”, “fica com fome”, “pega os garfos”, “come”, “solta os garfos” e então “pensa” novamente. São necessários dois garfos para um filósofo comer.

Abaixo estão implementadas a rotina pega_garfos (chamada pelo filósofo f quando ele fica com fome) e a rotina solta_garfos (chamada pelo filósofo f quando ele termina de comer). Os filósofos são numerados de 0 a 4. A rotina esquerda(f) retorna o número do filósofo a esquerda do filósofo f. De forma semelhante, a rotina direita(f) retorna o número do filósofo a direita do filósofo f.

Análise o código abaixo e procure responder as questões abaixo, justificando através de seqüências de eventos:

- Dois filósofos vizinhos podem comer simultaneamente ?
- Todos os filósofos podem ficar bloqueados com fome ao mesmo tempo ?
- Um filósofo pode ficar em postergação indefinida sem que todo o sistema pare ?

```

#define PENSA 0
#define FOME 1
#define COME 2
int estado[5] = { PENSA, PENSA, PENSA, PENSA, PENSA };
semaphore esperando[5] = { 0, 0, 0, 0, 0 };
semaphore exclusivo = 1;

```

<pre> void pega_garfos(int f) { P(exclusivo); if(estado[esquerda(f)] == PENSA && estado[direita(f)] == PENSA) { estado[f] = COME; V(exclusivo); } else { estado[f] = FOME; V(exclusivo); P(esperando[f]); } } </pre>	<pre> void solta_garfos(int f) { P(exclusivo); estado[f] = PENSA; if(estado[esquerda(f)] == FOME && estado[esquerda(esquerda(f)) == PENSA) { estado[esquerda(f)] = COME; V(esperando(esquerda(f)); } if(estado[direita(f)] == FOME && estado[direita(direita(f)) == PENSA) { estado[direita(f)] = COME; V(esperando(direita(f)); } V(exclusivo); } </pre>
--	--

5) O problema dos leitores/escritores consiste de um texto que pode ser lido ou escrito por vários processos. Considerando o código abaixo, responda justificando:

- e) É possível vários leitores lerem ao mesmo tempo ?
- f) É possível vários escritores escreverem ao mesmo tempo ?
- g) É possível um leitor e um escritor acessarem ao mesmo tempo ?
- h) É possível postergação indefinida de um escritor ?

```

int nl = 0;
semaphore tipo = 1;
semaphore exclusivo = 1;

```

<pre> void leitor(void) { ... P(exclusivo); if(nl > 0) ++ nl; else{ P(tipo); nl = 1; } V(exclusivo); Acessa o texto P(exclusivo); -- nl; if(nl == 0) V(tipo); V(exclusivo); ... } </pre>	<pre> void escritor(void) { ... P(exclusivo); Acessa o texto V(exclusivo); ... } </pre>
--	--

6) Usando **semáforos**, implemente rotinas cujo comportamento seja similar ao das funções wait e signal vistas em aula (pthreads) e descritas abaixo. Crie tantos semáforos e variáveis auxiliares do tipo inteiro quantos julgar necessário. Explique a lógica da solução.

WAIT(VC): Sempre bloqueia o processo até alguém executar um SIGNAL.

SIGNAL(VC): Se existe alguém bloqueado em VC, acorda apenas 1 deles.

Obs: O que é pedido são dois trechos de código em C, tal que um funcione como se fosse o “wait” e o outro funcione como se fosse o “signal”. Neste código em C, para obter atomicidade e bloquear tarefas, devem ser usados semáforos.

7) Em um programa concorrente existem N processos trabalhadores e um único processo finalizador. O programa é tal que o processo finalizador precisa esperar todos os processos trabalhadores terminarem para então ele finalizar o programa. Este tipo de sincronização é chamada de “barreira” e é típica da programação em máquinas paralelas. O valor de N é conhecido.

Implemente uma solução para o problema da barreira usando semáforos. A solução consiste de duas rotinas:

```
/* processo trabalhador “meuPid” informa que acabou sua parte do serviço */  
void acabei( int meuPid);
```

```
/* processo finalizador fica bloqueado até todos os trabalhadores acabarem o serviço */  
void espera_todos( void );
```

Esqueleto do código dos processos:

<pre>Void trabalhador(int eu) { ... acabei(eu); ... }</pre>	<pre>void finalizador(void) { ... espera_todos(); ... }</pre>
--	---

8) Questão idêntica a anterior, mas agora é obrigatório inicializar todos os semáforos com 1.

9) Em muitos sistemas operacionais existe uma chamada de sistema **yield()**, cujo efeito é passar o processo chamador para o final da fila de aptos. Suponha que um dado kernel escalone o processador utilizando fatias de tempo e suporte a chamada **yield()**. Suponha ainda que, em todas as aplicações, o tempo necessário para executar uma seção crítica seja menor que a duração de uma fatia de tempo. Para este caso em particular, buscando resolver o problema da seção crítica, o Bruno propôs o seguinte mecanismo de sincronização:

- Imediatamente antes de entrar na seção crítica, o processo chama **yield()**;
- Ao sair da seção crítica, o processo não faz nada.

Responda as questões abaixo, sempre justificando sua resposta com um exemplo.

- (a) Esta solução oferece exclusão mútua ?
- (b) Esta solução apresenta a possibilidade de postergação indefinida ?
- (c) Esta solução funciona para mais de dois processos ?
- (d) Esta solução apresenta busy-waiting ?

10) Considere uma relação produtor/consumidor onde vários produtores e vários consumidores acessam o mesmo buffer circular. O buffer circular possui capacidade para N dados (por exemplo, doubles). Cada produtor sempre deposita no buffer apenas 1 dado de cada vez. Entretanto, cada consumidor sempre retira do buffer 3 dados de cada vez. Além disto, o consumidor precisa retirar obrigatoriamente do buffer 3 dados que tenham sido depositados em seqüência pelos produtores. Crie uma solução para este problema utilizando semáforos. Tanto produtor quanto consumidor devem ficar bloqueados caso o buffer não tenha como atender o pedido imediatamente. Soluções com busy-waiting não são aceitáveis.

PTHREADS

11) Em um programa concorrente existem N processos trabalhadores e um único processo finalizador. O programa é tal que o processo finalizador precisa esperar todos os processos trabalhadores terminarem para então ele finalizar o programa. Este tipo de sincronização é chamada de “barreira” e é típica da programação em máquinas paralelas.

Implemente uma solução para o problema da barreira usando mutexes e variáveis condição da biblioteca das pthreads. A solução consiste de 2 rotinas:

```
/* processo trabalhador “meuPid” informa que acabou sua parte do serviço */  
void acabei (int meuPid);
```

```
/* processo finalizador fica bloqueado até todos os trabalhadores acabarem o serviço */  
void espera_todos( void );
```

Dicas sobre o Posix:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;  
pthread_mutex_lock( &m );  
pthread_mutex_unlock( &m );
```

```
pthread_cond_t vc = PTHREAD_COND_INITIALIZER;  
pthread_cond_wait( &vc , &m );  
pthread_cond_signal( &vc );
```

12) Suponha a existência de um texto que é acessado por vários processos Leitores e um processo Escritor. Quando o Escritor está acessando, nenhum Leitor pode ler o texto, pois poderia acessar uma versão inconsistente. Por outro lado, é permitido que vários Leitores acessem o texto simultaneamente. Implemente a solução descrita abaixo, usando os recursos da biblioteca Pthreads. Não é necessário criar as threads, apenas mostrar o código de acesso ao texto.

Leitor:

- Se número de leitores acessando for maior que zero
 - Então Incrementa número de leitores acessando
 - Senão Se escritor acessando
 - Então Espera escritor sair
 - Senão Faz Leitores acessando igual a 1
- Acessa o texto
- Decrementa número de leitores acessando
- Se número de leitores acessando for igual a zero
 - Então Libera o texto

Escritor: Se número de leitores acessando for maior que zero
 Então Espera ficar livre
 Senão Indica escritor acessando
 Acessa o texto
 Libera o texto

Dicas sobre o Posix:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;  
pthread_mutex_lock( &m );  
pthread_mutex_unlock( &m );
```

```
pthread_cond_t vc = PTHREAD_COND_INITIALIZER;  
pthread_cond_wait( &vc , &m );  
pthread_cond_signal( &vc );
```

13) Usando os mecanismos "mutex" e "variáveis condição" do Posix, implemente as operações P e V de um semáforo. Cada uma dessas operações deverá ser substituída por um código C com semântica similar. Lembre-se que estas operações devem ser atômicas. O valor inicial do semáforo é 1.

Dicas sobre o Posix:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;  
pthread_mutex_lock( &m );  
pthread_mutex_unlock( &m );
```

```
pthread_cond_t vc = PTHREAD_COND_INITIALIZER;  
pthread_cond_wait( &vc , &m );  
pthread_cond_signal( &vc );
```

14) Considere a situação de uma estrada que possui uma ponte cuja largura só comporta um carro. A estrada como um todo possui uma pista em cada sentido, mas a ponte possui uma única pista. Carros vindos do norte e carros vindos do sul chegam a ponte com uma só pista e precisam atravessá-la. Os que viajam no mesmo sentido podem atravessar a ponte ao mesmo tempo, mas os que viajam em sentido oposto precisam esperar que a ponte fique livre.

Modele os carros como threads e a ponte como um recurso, o qual pode ser utilizado simultaneamente por carros que travegam no mesmo sentido, mas nunca por carros em sentidos opostos. Implemente uma solução para este problema usando mutexes e variáveis condição da biblioteca das pthreads. A solução consiste de 2 rotinas, como mostrado abaixo:

```
/* função chamada pela thread carro para solicitar o acesso a ponte, informando o seu sentido  
(NORTE ou SUL), ao retornar desta função a thread está autorizada a entrar na ponte */  
void pede_para_entrar (int sentido);
```

```
/* função chamada pela thread carro para avisar que saiu da ponte, possivelmente liberando  
processos do outro sentido */  
void avisa_que_saiu (int sentido);
```

Dicas sobre o Posix:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;  
pthread_mutex_lock( &m );  
pthread_mutex_unlock( &m );
```

```
pthread_cond_t vc = PTHREAD_COND_INITIALIZER;  
pthread_cond_wait( &vc , &m );  
pthread_cond_signal( &vc );
```

15) Considere a situação de uma parada de onibus que funciona no estilo de lotação. Os passageiros formam uma fila dentro da estação (FIFO). Quando chega o onibus com N lugares, os passageiros embarcam. Se o onibus lotou, ele parte em sua jornada. Senão, ele fica parado até que cheguem passageiros em número suficiente para lotar o onibus, quando então ele parte. Quando um passageiro chega na estação e o onibus está parado esperando passageiros, ele embarca imediatamente. Senão ele fica esperando até o ônibus chegar.

Modele as pessoas e o onibus como threads e a estação como um monitor. Implemente este sistema usando mutexes e variáveis condição da biblioteca das pthreads. A solução consiste de um monitor com 2 rotinas de entrada, como mostrado abaixo:

```
/* thread pessoa chega na estação para embarcar no onibus, fica bloqueada até seu embarque no  
ônibus ser efetuado */
```

```
void embarca_no_onibus ( );
```

```
/* thread ônibus chega na estação para embarcar passageiros até sua lotação completa, fica  
bloqueado até ficar com a lotação completa */
```

```
void espera_encher_onibus ( );
```

Esqueleto do código das threads:

<pre>void pessoa(void) { ... Embarca_no_onibus(); ... }</pre>	<pre>void onibus(void) { ... Espera_encher_onibus(); ... }</pre>
---	--

Dicas sobre o Posix:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;  
pthread_mutex_lock( &m );  
pthread_mutex_unlock( &m );
```

```
pthread_cond_t vc = PTHREAD_COND_INITIALIZER;  
pthread_cond_wait( &vc , &m );  
pthread_cond_signal( &vc );
```

16) Implemente na forma de monitor, usando variáveis mutex e variáveis condição, um esquema de buffer duplo. O monitor possui duas rotinas de acesso, uma para inserção de um número tipo double, e outra para a leitura de um array com 100 doubles previamente escritos. Internamente ao monitor existem 2 buffers, cada um deles comporta 100 doubles. A cada momento existe 1 buffer em uso, enquanto o outro está ou completamente vazio ou completamente cheio. Os dados devem ser lidos na mesma ordem na qual foram escritos.

```
void escreve_double( double d);
```

A escrita do double sempre ocorre no buffer em uso. Quando o buffer em uso fica cheio, o outro buffer passa a ser o buffer em uso, e assim por diante. No caso dos dois buffers estarem cheios, a thread que deseja escrever fica bloqueada até que uma leitura aconteça, e libere um buffer.

```
void le_array( double *pd);
```

A leitura somente acontece se um buffer estiver cheio, o buffer é lido por inteiro de uma só vez, e fica liberado para novas escritas. Caso a thread tente ler, mas nenhum dos dois buffers está cheio no momento, ela fica bloqueada até que o buffer em uso fique cheio.

Esqueleto do código das threads:

<pre>void thread_coloca(void) { ... escreve_double(dado); ... }</pre>	<pre>void thread_tira(void) { ... le_array(array); ... }</pre>
---	--

Dicas sobre o Posix:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock( &m );
pthread_mutex_unlock( &m );
```

```
pthread_cond_t vc = PTHREAD_COND_INITIALIZER;
pthread_cond_wait( &vc , &m );
pthread_cond_signal( &vc );
```

17) No problema dos leitores e escritores, é permitido que vários leitores possam acessar os dados para leitura simultânea, mas cada escritor precisa acessar os dados exclusivamente para efetuar a escrita. Abaixo está proposta uma solução para o problema, usando mutex e variáveis condição da biblioteca pthreads. É suposto que cada leitor chama “pede_ler” antes da leitura e chama “fim_ler” após a leitura. Da mesma forma, cada escritor chama “pede_escrever” antes da escrita e “fim_escrever” depois da escrita.

(a) Mostre através de um exemplo que a solução abaixo não funciona;

(b) Altere a solução para que ela funcione, mesmo que podendo favorecer um tipo de processo, mostre através de um exemplo de execução com chegada de: leitor, escritor, leitor, escritor.

```
Mutex em;
Cond liv;
Int n_lei = 0;
Int n_esc = 0;
```

<pre>void pede_ler(void) { Lock(&em); If(n_esc > 0) Wait(&liv, &em); ++n_lei; Unlock(&em); }</pre>	<pre>void pede_escrever(void) { Lock(&em); If(n_lei > 0) Wait(&liv, &em); ++n_esc; Unlock(&em); }</pre>
--	---


```

void fim_ler(void)
{
    Lock(&em);
    --n_lei;
    Signal(&liv);
    Unlock(&em);
}

```

```

void fim_escrever(void)
{
    Lock(&em);
    --n_esc;
    Signal(&liv);
    Unlock(&em);
}

```

18) Considere uma relação produtor/consumidor onde existem 2 tipos de produtores, produtor tipo A e produtor tipo B, e um único tipo de consumidor. Existem várias threads produtoras e consumidoras. Implemente um monitor utilizando as funções da biblioteca de pthreads, tal que existam 3 chamadas possíveis:

- void insere_tipo_A(double x)
- void insere_tipo_B(double x)
- double retira(void)

Dentro do monitor existem 2 buffers circulares, um para dados do tipo A e outro para dados do tipo B. Um produtor fica bloqueado quando o seu buffer está cheio. O consumidor sempre retira um item de cada buffer. Desta forma, o consumidor deve ficar bloqueado caso qualquer um dos buffers esteja vazio. A função retira retorna o produto dos dois valores (tipo A e tipo B) consumidos. Soluções com busy-waiting não são aceitáveis.

19) No código mostrado abaixo busca-se implementar a semântica de semáforos (operações P e V) através de duas funções que utilizam mutexes para sincronizar threads. Suponha que existam 3 threads A, B e C as quais chamam a função “p” e que existam também outras 3 threads X, Y e Z as quais chamam a função “V”. O valor inicial do semáforo hipotético é zero, indicado pela variável “cont”. As operações lock/unlock sobre o mutex funcionam normalmente.

Discuta sobre a corretude da implementação abaixo. Considere as situações quando são executadas operações V sobre o semáforo, estando bloqueadas no semáforo um total de 0, 1, 2 e 3 threads.

```

mutex em;
mutex bloqueia;
int cont = 0;

```

```

void p(void)
{
    Lock(&em);
    cont = cont - 1;
    If( cont < 0 )
    {
        Unlock(&em);
        Lock(&bloqueia);
    }
    Else
    {
        Unlock(&em);
    }
}

```

```

void v(void)
{
    Lock(&em);
    cont = cont + 1;
    If( cont <= 0 )
    {
        Unlock(&bloqueia);
    }
    Unlock(&em);
}

```

20) Considere uma relação produtor/consumidor onde existem 2 tipos de produtores (produtor tipo A e produtor tipo B), e dois tipos de consumidores (consumidor de produto e consumidor tipo A). Existem várias threads produtoras e consumidoras. Implemente um monitor utilizando as funções da biblioteca de pthreads, tal que existam 4 chamadas possíveis:

- void insere_tipo_A(double x)
- void insere_tipo_B(double x)
- double retira_produto(void)
- double retira_A(void)

Dentro do monitor existem 2 buffers circulares, um para dados do tipo A e outro para dados do tipo B. Um produtor fica bloqueado quando o seu buffer está cheio. O consumidor de produto sempre retira um item de cada buffer. Desta forma, este consumidor deve ficar bloqueado caso qualquer um dos buffers esteja vazio. A função retira_produto retorna o produto dos dois valores (tipo A e tipo B) consumidos. O outro consumidor só retira elementos do buffer A. Soluções com busy-waiting não são aceitáveis.

21) Considere um sistema de medição onde existem duas threads sensoras. Cada thread sensora faz medições em vários pontos da planta, gerando uma “struct C” com todos os dados lidos por ela em uma dada medição. As threads sensoras são identificadas como tipo A e tipo B.

Existe neste sistema também uma série de threads que utilizam os dados medidos para vários propósitos, tais como alimentar sinótico, gerar históricos, etc. Essas threads consultam sempre a última medição disponível de um dado tipo.

Finalmente, existe neste sistema também uma thread de alarme, a qual deve ser ativada na ocorrência de uma combinação complexa de valores nos dados medidos dos dois tipos simultaneamente. Esta condição é indicada pela função auxiliar “int situacao_alarme(void)”, a qual retorna “1” quando a situação de alarme está ocorrendo. Esta função auxiliar já está implementada.

Implemente um monitor utilizando as funções da biblioteca de pthreads, sem busy-waiting, tal que existam 3 chamadas possíveis:

```
Struct dados_medidos_t {  
    ...  
    ...  
};
```

```
#define TIPO_A 0  
#define TIPO_B 1
```

- void insere_medicao(int tipo, struct dados_medidos_t xxx)
- struct dados_medidos_t consulta_medicao(int tipo)
- void espera_alarme(void)

Dentro do monitor são mantidas apenas a última medição disponível para cada tipo, ou seja, existe uma estrutura para a medição mais recente do tipo A e outra para a medição mais recente do tipo B.

A função consulta_medicao() apenas informa o valor atual daquele tipo de medição sem consumi-lo (apaga-lo) do monitor. No caso de não haver ainda nenhuma medição do tipo desejado no monitor, a thread fica bloqueada até uma medição do tipo desejado ser incluída no monitor.

A função `espera_alarme()` bloqueia a thread chamadora até que a situação de alarme seja detectada (torne-se verdadeira).

22) Implemente na forma de monitor, usando variáveis mutex e variáveis condição, uma biblioteca para troca de mensagens através de caixa postal. O monitor que implementa a caixa postal possui duas rotinas de acesso, uma rotina para o envio de uma mensagem e outra rotina para a recepção de uma mensagem. Quando uma thread tenta receber uma mensagem mas nenhuma mensagem foi ainda colocada na caixa postal, a thread fica bloqueada. A caixa postal pode bufferizar até 1K bytes de mensagens. Quando uma thread tenta colocar uma mensagem na caixa postal lotada, ela recebe um retorno -1, mas não fica bloqueada. Mensagens são do tipo string de caracteres e possuem tamanho variado. As mensagens devem ser entregues na mesma ordem na qual foram enviadas. Implemente a solução para o caso de uma única caixa postal.

Rotinas a serem implementadas:

```
int envia_mensagem( char *msg);  
void recebe_mensagem( char *msg);
```

Dicas sobre o Posix:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;  
pthread_mutex_lock( &m );  
pthread_mutex_unlock( &m );
```

```
pthread_cond_t vc = PTHREAD_COND_INITIALIZER;  
pthread_cond_wait( &vc , &m );  
pthread_cond_signal( &vc );
```

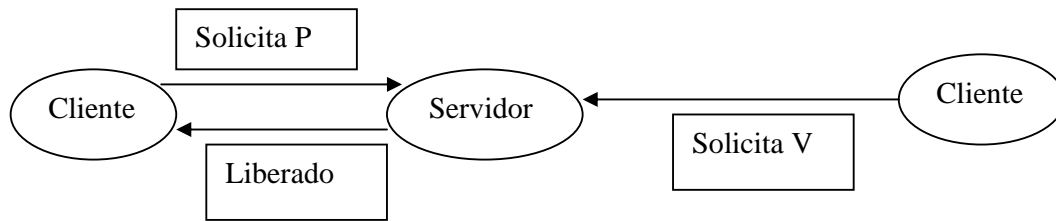
MENSAGENS

23) Em um sistema que suporta programação concorrente apenas através da troca de mensagens, será criado um Servidor para controlar o uso das portas seriais. Quando um processo Cliente deseja usar uma porta serial, ele envia uma mensagem “Aloca” para o Servidor. Existem N portas seriais, todas equivalentes, mas cada uma pode ser usada somente por um Cliente de cada vez. O Servidor informa ao Cliente a porta que ele vai usar através da mensagem “Porta p”. Ao concluir o uso, o Cliente envia para o Servidor a mensagem “Libera p”. Suponha que exista mais do que N processos Clientes. Mostre o algoritmo do Servidor, em português estruturado. Apresentar a solução como foi feito para o Servidor de Buffer visto em aula. Supor “receive” bloqueante.

24) Em um sistema que suporta programação concorrente apenas através da troca de mensagens, será criado um Servidor de Semáforo. Suponha que o processo S será o servidor, implementando a funcionalidade de um único semáforo inicializado com 1. O protocolo que permite a um processo cliente executar operações P e V é mostrado abaixo. Mostre o algoritmo do processo S, em português estruturado. Apresentar a solução como foi feito para o Servidor de Buffer visto em aula. Supor “receive” bloqueante.

P: --valor
Se valor < 0 então bloqueia quem executou o P

V: ++valor
Acorda o primeiro da fila, se houver alguém bloqueado

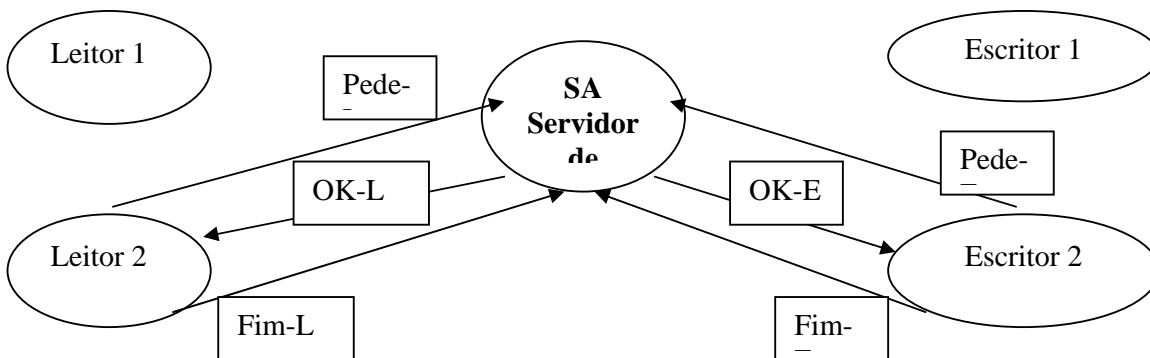


25) Em um sistema que suporta programação concorrente apenas através da troca de mensagens, será criado um **Servidor de Acesso** para controlar o acesso de vários processos a um dado arquivo. Os processos clientes podem ser do tipo Leitor (apenas lê o arquivo) ou do tipo Escritor (pode alterar o arquivo). O controle de acesso é tal que vários processos podem ler simultaneamente do arquivo, mas cada escritor deve acessar sozinho o arquivo.

- O protocolo entre Leitores/Escritores e o Servidor de Acesso é mostrado na figura.
- O esqueleto do código dos leitores e dos escritores é mostrado abaixo.

Mostre o algoritmo do Servidor de Acesso em português estruturado como feito em aula. Supor que o Servidor de Acesso é monothread e a chamada “receive()” é bloqueante.

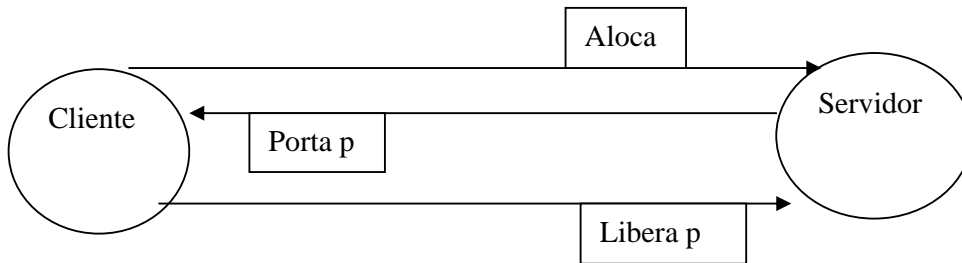
Em caso de disputa, o Servidor de Acesso poderá dar preferência para um ou outro tipo de cliente, mesmo que isto possa causar postergação indefinida do tipo de cliente preterido.



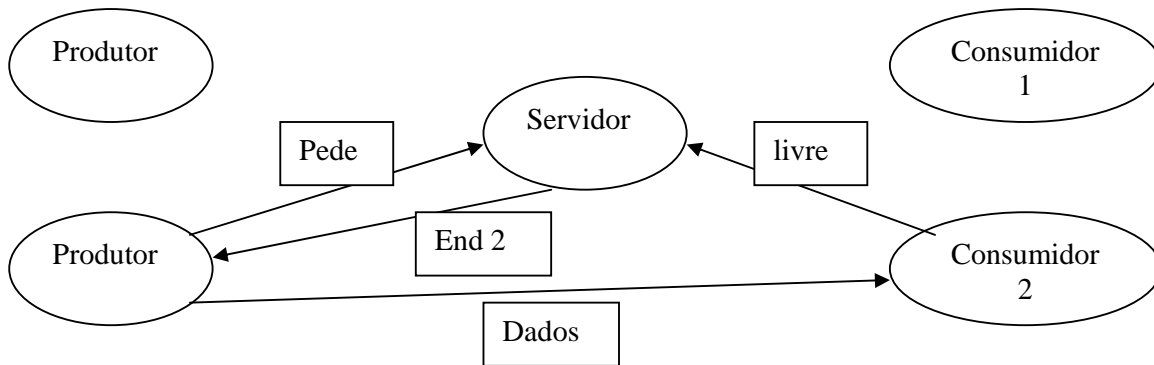
<pre> void leitor(int meu_id) { While (true) { ... Send(SA, "pede-L"); Receive(SA, "ok-L"); ... lê ... Send(SA, "Fim-L"); ... } } </pre>	<pre> void escritor(int meu_id) { While (true) { ... Send(SA, "pede-E"); Receive(SA, "ok-E"); ... escreve ... Send(SA, "Fim-E"); ... } } </pre>
---	--

26) Em um sistema que suporta programação concorrente apenas através da troca de mensagens, será criado um Servidor para controlar o uso das portas seriais. Quando um processo Cliente deseja usar uma porta serial, ele envia uma mensagem “Aloca” para o Servidor. Existem N portas seriais,

todas equivalentes, mas cada uma pode ser usada somente por um Cliente de cada vez. O Servidor informa ao Cliente a porta que ele vai usar através da mensagem “Porta p”. Ao concluir o uso, o Cliente envia para o Servidor a mensagem “Libera p”. Suponha que exista mais do que N processos Clientes. Mostre o algoritmo do Servidor, em português estruturado. Apresentar a solução como foi feito para o Servidor de Buffer visto em aula. Supor “receive” bloqueante.

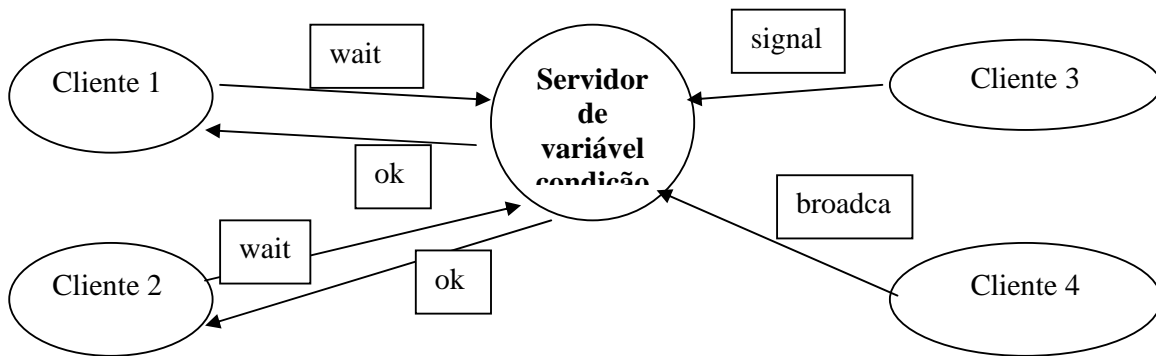


27) Em um sistema que suporta programação concorrente apenas através da troca de mensagens, será criado um Servidor para controlar o envio de dados entre produtores e consumidores. Os dados serão enviados diretamente de um Produtor para um Consumidor livre, mas o Produtor deve consultar o Servidor para obter o número do Consumidor que ele deverá usar. O Servidor deve manter uma lista com os números dos Consumidores livres no momento, e informar um desses números quando um Produtor solicita. Após receber e processar os dados, o Consumidor volta a informar que está livre para o Servidor. Mostre o algoritmo do Servidor, em português estruturado. Apresentar a solução como foi feito para o Servidor de Buffer visto em aula. Supor “receive” bloqueante.

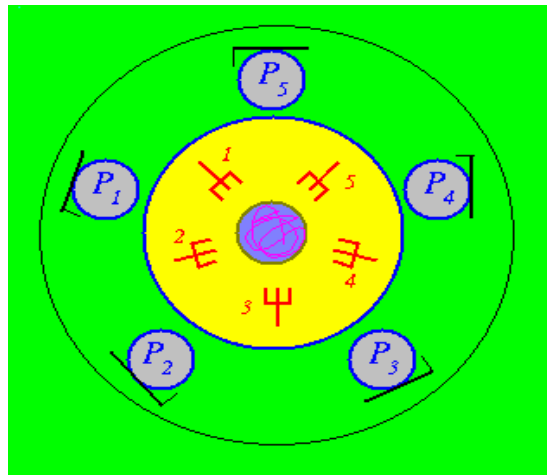


28) Em um sistema que suporta programação concorrente apenas através da troca de mensagens, será criado um **Servidor de Variável Condição** para implementar o comportamento típico de uma variável condição das pthreads. Os processos clientes podem enviar para o servidor as seguintes mensagens: “wait” (deixa o cliente esperando em uma fila do servidor por uma resposta “ok”), “signal” (faz o servidor mandar uma resposta “ok” apenas para o primeiro cliente da fila de espera) e “broadcast” (faz o servidor mandar uma resposta “ok” para todos os clientes que estavam em sua fila de espera).

Mostre o algoritmo do Servidor de Variável Condição em Português Estruturado como feito em aula. Supor que o Servidor é monothread e a chamada “receive()” é bloqueante.



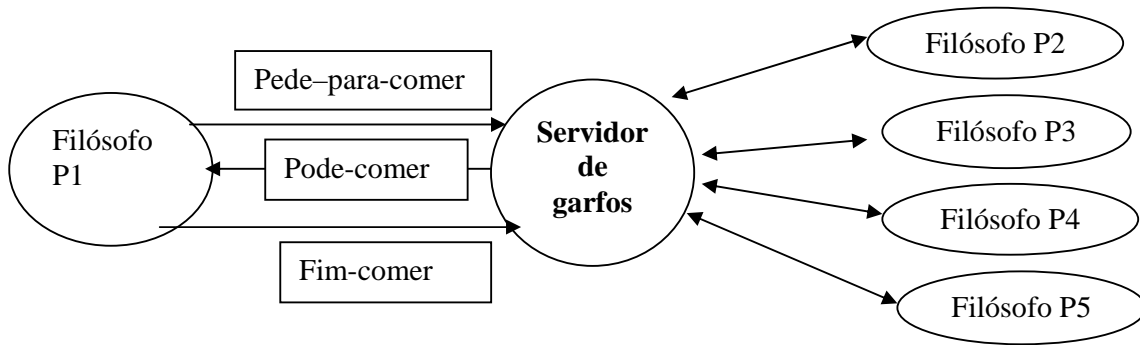
29) Considere o problema dos filósofos jantadores, onde 5 filósofos estão sentados em torno de uma mesa circular. A frente de cada um existe um prato e entre cada 2 filósofos existe 1 garfo. Cada filósofo vive um eterno ciclo de “pensa”, “fica com fome”, “pega os garfos”, “come”, “solta os garfos” e então “pensa” novamente. São necessários dois garfos para um filósofo comer. Por exemplo, o filósofo P4 da figura precisa dos garfos 4 e 5 para comer. Enquanto ele come, os filósofos P3 e P5 estão impedidos de comer.



Em um sistema que suporta programação concorrente apenas através da troca de mensagens, deve ser criado um **Servidor de Garfos para os Filósofos Jantadores**, para gerenciar o uso de garfos por parte dos filósofos. Cada filósofo, quando fica com fome, envia a mensagem “pede-para-comer” ao servidor. Quando aquele filósofo estiver autorizado a comer, o servidor envia para ele a mensagem “pode-comer”. Finalmente, quando o filósofo termina de comer, ele avisa ao servidor através da mensagem “fim-comer”.

O servidor deve tomar cuidado para evitar que dois filósofos vizinhos comam simultaneamente. Também deve evitar que todos os filósofos fiquem bloqueados com fome ao mesmo tempo. Uma solução simples é atender aos filósofos por ordem de chegada, na medida que a disponibilidade de garfos permitir.

Mostre o algoritmo do Servidor de Garfos em Português Estruturado como feito em aula. Supor que o Servidor é monothread e a chamada “receive()” é bloqueante.

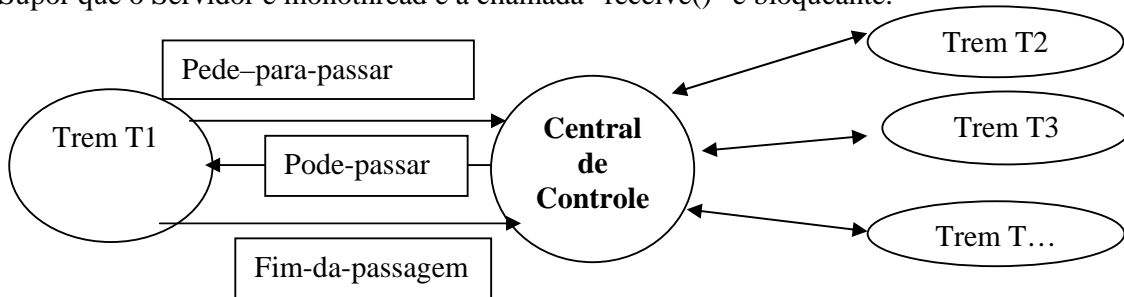


30) Considere um cruzamento ferroviário automatizado, onde cada trem possui um computador abordo e no cruzamento também existe uma central de controle. Os computadores nos trens conversam apenas com a Central de Controle e não entre si. A comunicação ocorre no padrão cliente-servidor, sendo a Central de Controle o servidor e os computadores abordo dos trens os clientes.

Em um sistema que suporta programação concorrente apenas através da troca de mensagens, deve ser descrito o algoritmo da Central de Controle, para gerenciar a passagem dos trens pelo cruzamento. Cada trem, ao aproximar-se do cruzamento, envia a mensagem “pede-para-passar” ao servidor. Quando aquele trem estiver autorizado a passar pelo cruzamento, o servidor envia para ele a mensagem “pode-passar”. Finalmente, quando o trem sai do cruzamento, ele avisa ao servidor através da mensagem “fim-da-passagem”.

O servidor não pode permitir que dois trens estejam no cruzamento ao mesmo tempo. A ordem de atendimento dos trens é FIFO, mas não é possível prever quantos trens irão pedir passagem ao mesmo tempo.

Mostre o algoritmo da Central de Controle em Português Estruturado como feito em aula. Supor que o Servidor é monothread e a chamada “receive()” é bloqueante.



TEMPO REAL

31) Determine se o conjunto de tarefas abaixo é escalonável com deadline monotônico (prioridades preemptivas), aplicando a análise para $D \leq P$, vista em aula.

T1:	C1=2	P1=5	D1=5
T2:	C2=5	P2=25	D2=10
T3:	C3=10	P3=25	D3=25

32) Considerando o conjunto de tarefas abaixo, é possível criar uma escala de tempo preemptiva para atender os deadlines solicitados ? E uma não preemptiva ? Justifique.

T1:	C1=2	P1=5	D1=5
T2:	C2=5	P2=25	D2=10
T3:	C3=10	P3=25	D3=25

33) Determine se o conjunto de tarefas abaixo é escalonável com deadline monotônico, aplicando a análise para $D \leq P$, vista em aula.

T1:	J1=0	C1=2	P1=5	D1=5	B1 = 0
T2:	J2=0	C2=5	P2=30	D2=10	B2 = 0
T3:	J3=0	C3=10	P3=25	D3=25	B3 = 0

$$W_i = C_i + B_i + \sum_{j \in HP(i)} \left\lceil \frac{J_j + W_i}{P_j} \right\rceil \times C_j \quad R_i = J_i + W_i$$

34) Determine se o conjunto de tarefas abaixo é escalonável, aplicando a análise baseada em tempo de resposta vista em aula, quando:

- (a) prioridades são atribuídas conforme o deadline monotônico;
- (b) prioridades são atribuídas conforme o taxa monotônico (rate monotonic);

T1:	C1=3	P1=20	D1=6
T2:	C2=3	P2=15	D2=7
T3:	C3=4	P3=9	D3=8

35) Considerando o mesmo conjunto de tarefas, é possível criar uma escala de tempo não preemptiva para atender os deadlines solicitados ? Justifique tentando criar uma escala do tipo usada por executivo cíclico.

36) Ilustre através de exemplos como as características abaixo do sistema operacional podem aumentar o tempo de resposta de uma tarefa com alta prioridade. Para cada exemplo, desenhe um diagrama de tempo para mostrar o impacto sobre a tarefa de alta prioridade.

- (a) Interrupções desabilitadas.
- (b) Kernel não é preemptivo.

37) Indique dois aspectos do projeto de um sistema operacional que afetam o comportamento temporal de uma aplicação, atrapalhando o atendimento dos requisitos temporais.

38) Eventos são coletados em uma rede composta por 4 computadores, e associados com a hora local do computador que detecta cada evento. Para que os mesmos possam ser analisados em conjunto, é necessário um erro máximo entre relógios de 100 milissegundos. Os cristais usados nesses computadores possuem uma precisão de 10^{-5} em relação a sua frequência nominal, para mais ou para menos.

Supondo que pode existir um erro inicial de 40 milissegundos entre quaisquer dois relógios da rede, calcule quanto tempo vai levar até que os dois relógios com maior erro entre eles apresentem um erro de 100 milissegundos, supondo sempre o pior caso.

39) Descreva a diferença entre deadlines hard, firm e soft, e indique um exemplo para cada um deles.

40) Indique dois aspectos do projeto de um sistema operacional que afeta o comportamento temporal de uma aplicação.

41) Qual a diferença entre tempo real crítico e tempo real brando ? Forneça também uma aplicação exemplo para cada tipo.

42) Determine se o conjunto de tarefas abaixo é escalonável se prioridades forem atribuídas segundo o deadline monotônico, aplicando a análise para $D \leq P$, vista em aula. Considere que existe uma seção crítica entre T1 e T2, em função de uma tabela acessada pelas duas tarefas. A tarefa T1 permanece 1 unidade de tempo acessando a tabela, enquanto a tarefa T2 permanece 2 unidades de tempo acessando a mesma tabela.

T1:	J1=0	C1=2	P1=5	D1=5
T2:	J2=0	C2=5	P2=30	D2=10
T3:	J3=0	C3=10	P3=25	D3=25

$$W_i = C_i + B_i + \sum_{j \in HP(i)} \left\lceil \frac{J_j + W_i}{P_j} \right\rceil \times C_j \quad R_i = J_i + W_i$$

43) Considerando o mesmo conjunto de tarefas, é possível criar uma escala de tempo não preemptiva para atender os deadlines solicitados ? Justifique tentando criar uma escala do tipo usada por executivo cíclico.

44) Considere um kernel preemptivo com as seguintes características:

- Interrupções do timer acontecem a cada 10 ms;
- O tratador de interrupções do timer demora 1 ms para executar;
- Dentro do kernel as interrupções ficam desabilitadas no máximo por 2 ms de cada vez;
- A carga do contexto de uma tarefa demora 0,1 ms e o salvamento do contexto de uma tarefa demora 0,2 ms.

Supondo que a tarefa da aplicação de mais alta prioridade execute a cada ativação um algoritmo que demora (tempo de computação) no pior caso 200 ms, qual será o seu tempo máximo de resposta ? Justifique cada termo da equação usada no cálculo.

45) Considerando um sistema de tempo real que trabalha com escalonamento preemptivo e prioridades fixas, como apareceria para uma tarefa de alta prioridade, em termos de impacto sobre o seu tempo de resposta, os seguintes fatores:

(a) Tratador das interrupções de hardware do timer, tratador demora CT mS para executar e o timer está programado para ocorrer a cada PT mS.

(b) Kernel desabilita interrupções para executar operações críticas, o trecho mais longo de interrupções desabilitadas demora ID mS.

46) Eventos são coletados em uma rede composta por 4 computadores, e associados com a hora local do computador que detecta cada evento. Para que os mesmos possam ser analisados em conjunto, é necessário um erro máximo entre relógios de 100 milisegundos. Os cristais usados nesses computadores possuem uma precisão de 10 elevado a -5 em relação a sua frequência nominal, para mais ou para menos. Calcule quanto tempo vai levar até que os dois relógios com maior erro entre eles apresentem um erro de 100 milisegundos, supondo sempre o pior caso.

47) Considere uma aplicação composta por algumas tarefas de controle periódicas e algumas tarefas esporádicas associadas com alarmes. Compare a abordagem baseada em executivo cíclico com a abordagem baseada em prioridades com respeito ao escalonamento do sistema. Que fatores favorecem uma e outra ?

48) Determine se o conjunto de tarefas abaixo é escalonável com deadline monotônico (prioridades preemptivas), aplicando a análise para $D \leq P$, vista em aula.

T1:	C1=2	P1=5	D1=5
T2:	C2=6	P2=30	D2=10
T3:	C3=10	P3=30	D3=25

$$R_i = J_i + W_i \quad W_i = C_i + \sum_{j \in HP(i)} \left\lceil \frac{W_i + J_j}{P_j} \right\rceil \times C_j$$

49) Determine se o conjunto de tarefas abaixo é escalonável com deadline monotônico (prioridades preemptivas), aplicando a análise para $D \leq P$, vista em aula. Existe uma seção crítica protegida por mutex comum entre as tarefas T2 e T3, a qual demora 1 unidade de tempo para ser executada.

T1:	C1=2	P1=5	D1=5
	T2:	C2=5	P2=20
	T3:	C3=3	P3=30
			D2=16
			D3=30

$$R_i = C_i + B_i + \sum_{j \in HP(i)} \left\lceil \frac{R_i}{P_j} \right\rceil \times C_j$$